# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# THESIS

## DATA TRANSFORMATION IN A THREE DIMENSIONAL INTEGRATED CIRCUIT IMPLEMENTATION

by

Dimitrios Megas and Kleber Leandro Pizolato Someira

March 2012

| | |
|---|---|
| Thesis Advisor: | Ted Huffmire |
| Second Reader: | Timothy E. Levin |

**Approved for public release; distribution is unlimited**

THIS PAGE INTENTIONALLY LEFT BLANK

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved OMB No. 0704-0188* |

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

| | | |
|---|---|---|
| **1. AGENCY USE ONLY *(Leave blank)*** | **2. REPORT DATE**<br>March 2012 | **3. REPORT TYPE AND DATES COVERED**<br>Master's Thesis |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** Data Transformation in a Three Dimensional Integrated Circuit Implementation | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** Dimitrios Megas, Kleber Leandro Pizolato Someira | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)**<br>N/A | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** |

**11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ______N/A______.

| | |
|---|---|
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT**<br>Approved for public release; distribution is unlimited | **12b. DISTRIBUTION CODE**<br>A |

**13. ABSTRACT (maximum 200 words)**

Three-dimensional integration is an emerging chip fabrication technique in which multiple integrated circuit dies are joined using conductive posts. 3D integration offers several performance and security advantages, including extremely high bandwidth between the two dies and the ability to augment a processor with a separate die housing custom security features. This thesis performs a feasibility and requirements analysis of a data transformation coprocessor in a three-dimensional integrated circuit. We propose a novel coprocessor architecture in which one layer (control layer) houses application-specific coprocessors for cryptography and compression, which provide acceleration for applications running on a general-purpose processor in another layer (computational layer).

The main application supported from our proposed 3DIC is the one that performs real-time trace collection, compresses the trace, and optionally encrypts the compressed trace, which protects the data from interception during transmission to permanent off-chip storage for offline program analysis.

Although we are not building a hardware device for simulation we present the architecture for a 3D data transformation processor and a rationale for each of the key design decisions, including a compression study that determined the optimal compression algorithm for a specific set of traces.

| | |
|---|---|
| **14. SUBJECT TERMS**<br>Three Dimensional Integrated Circuit, Compression, Cryptography, Coprocessor, Data Transformation, Control Plane, Computational Plane | **15. NUMBER OF PAGES**<br>225 |
| | **16. PRICE CODE** |

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT**<br>Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE**<br>Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT**<br>Unclassified | **20. LIMITATION OF ABSTRACT**<br>UU |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release; distribution is unlimited**

# DATA TRANSFORMATION IN A THREE DIMENSIONAL INTEGRATED CIRCUIT IMPLEMENTATION

Dimitrios Megas
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1998

Kleber Leandro Pizolato Someira
Lieutenant Commander, Brazilian Navy
B.S., Brazilian Naval Academy, 1998

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**March 2012**

Authors: Dimitrios Megas
Kleber Leandro Pizolato Someira

Approved by: Ted Huffmire
Thesis Advisor

Timothy E. Levin
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Three-dimensional integration is an emerging chip fabrication technique in which multiple integrated circuit dies are joined using conductive posts. 3D integration offers several performance and security advantages, including extremely high bandwidth between the two dies and the ability to augment a processor with a separate die housing custom security features. This thesis performs a feasibility and requirements analysis of a data transformation coprocessor in a three-dimensional integrated circuit. We propose a novel coprocessor architecture in which one layer (control layer) houses application-specific coprocessors for cryptography and compression, which provide acceleration for applications running on a general-purpose processor in another layer (computational layer).

The main application supported from our proposed 3DIC is the one that performs real-time trace collection, compresses the trace, and optionally encrypts the compressed trace, which protects the data from interception during transmission to permanent off-chip storage for offline program analysis.

Although we are not building a hardware device for simulation we present the architecture for a 3D data transformation processor and a rationale for each of the key design decisions, including a compression study that determined the optimal compression algorithm for a specific set of traces.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

# LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| 2D | Two Dimensional |
| 3D | Three Dimensional |
| 3DIC | Three-Dimensional Integrated Circuit |
| ABYSS | A Basic Yorktown Security System |
| AES | Advanced Encrypted Standard |
| ALU | Arithmetic Logic Unit |
| ASCII | American Standard Code for Information Interchange |
| ATUM | Address Tracing Using Microcode |
| AVIRIS | Airborne Vehicle, Infrared-Imaging Spectrometer |
| BTA | Branch-Target Address |
| BYU | Brigham Young University |
| CAM | Content-Address Memory |
| CBC | Cipher-Block Chaining |
| CFB | Cipher Feedback |
| CMP | Chemical–Mechanical Polishing |
| CPU | Central-Processor Unit |
| CTID | Core Thread Identification |
| CTR | Counter |
| d2d | Die to Die |
| DASC | Data-Address Stride Cache |
| DES | Data-Encryption Standard |
| DFCM | Differential-Finite Context Method |
| DFT | Design for Testability |

| | |
|---|---|
| DH | Diffie – Hellman |
| DLLHT | Dynamic – Literal Length Huffman Tree |
| DOHT | Dynamic – Offset Huffman Tree |
| DRAM | Dynamic, Random-Access Memory |
| ECB | Electronic Code Book |
| ECC | Elliptical Curve Cryptography |
| ETA | Exception-Handler Target Address |
| FCM | Finite Context Method |
| FIFO | First In, First Out |
| GA | Genetic Algorithm |
| GB | Giga Byte |
| HTML | Hypertext Markup Language |
| I/O | Input/Output |
| IC | Integrated Circuit |
| ILP | Integer Linear Programming |
| ISA | Instruction-Set Architectures |
| L | Length |
| LV | Last Value |
| MB | Mega Byte |
| MD | Message Digest |
| MLBS | Multilayer Buried Structure |
| MTF | Move to Front |
| NIST | National Institute of Standards and Technology |
| OFB | Output Feedback |
| PC | Program Counter |

|  |  |
|---|---|
| PDA | Personal Digital Assistant |
| RAM | Random-Access Memory |
| RLE | Run-Length Encoding |
| ROM | Read-Only Memory |
| RSA | Rivest – Shamir – Adleman |
| SA | Simulated Annealing |
| SA | Starting Address |
| SBC | Stream-Based Compression |
| SC | Sequential Counting |
| SHA | Secure-Hash Algorithm |
| SL | Stream Length |
| SLLHT | Static – Literal Length Huffman Tree |
| SOHT | Static – Offset Huffman Tree |
| SoC | System on Chip |
| SRAM | Static, Random-Access Memory |
| TA | Target Address |
| TAM | Test Access Mechanism |
| TLB | Translation Look-Aside Buffer |
| TRM | Tamper-Resistant Modules |
| TSV | Through-Silicon Via |
| UD | Uniquely Decipherable |
| VDC | Volts of Direct Current |

THIS PAGE INTENTIONALLY LEFT BLANK

# ACKNOWLEDGMENTS

The authors would like to express their sincere appreciation to the people listed below who have provided support and advice, and who made this thesis possible. They were there from its initial concept to its published reality.

From LT Dimitrios Megas H.N:

I would like to express my thanks to Professor Ted Huffmire and Mr Timothy Levin, my thesis advisors. Thank you for the constructive critiques, and helping me put my thoughts on paper. Your support and guidance throughout this process truly made my research a valuable experience. In order to express my gratitude to them I would like to share with them a quote by one of my ancestors:

"I am indebted to my father for living, but to my teacher for living well."

– Alexander the Great

I would like to thank from my deepest of my heart my thesis partner and great friend Kleber. His friendship is definitely one of the greatest gifts I earned here. I am wishing you all the luck and success both professional and personal.

I would also like to thank Hellenic Navy for providing me the opportunity to pursue my studies here in Monterey, and Greek taxpayers, especially this specific period for my country, without their sacrifices my studies would not be possible.

I have to thank my parents, my father Lampros, my mother Vasiliki, and my sister Eleni for everything they have done for me so far.

Last but not least I would like to thank my wife Vasiliki for her endless love and support. Thank you for persuading me to strive for excellence. This thesis is dedicated to my precious daughter Erato, my inspiration to my whole life, and to my beloved wife Vasilki.

From LCDR Kleber Pizolato, Brazilian Navy:

I would like to recognize Professors Ted Huffmire and Timothy Levin, for the guidance through this research work, always pointing the right direction to follow, and

making constructive comments. My belief in their solid knowledge made me improve the quality of this research. Not only for their orientation, but for the friendly relationship we developed, I give not only my acknowledgements, but also my Brazilian friendship.

Friendship is not enough to define the relation with my partner Dimitrios, for his contribution to my life during this period I will be always remember him as a brother. Wish you and your family the best in your lives.

Thanks to my Country and the Brazilian Navy for believing in my work and give me the opportunity to enhance my knowledge trough my studies.

I have to thank my parents Waldomiro and Neuza for the gift of life and for solid education. To my sister Kleisse, her friendship and persistence made me stronger during the difficult moments.

My special thank to my wife Vanessa, who is always beside me with her endless love. Your support and motivation were fundamental to the success of this journey, one more journey of our lives!

# I. INTRODUCTION

## A. MOTIVATION

Application-specific coprocessors, including those for cryptography and compression, can provide significant acceleration and power savings to programs requiring their services. While most coprocessors have traditionally been constructed as a separate chip connected to the main CPU over a relatively slow bus connection, 3D integration is an emerging technology that offers significant performance advantages and power savings over traditional systems that combine chips at the circuit board level. [1], [2], [3], [4]. With 3D integration, two or more dies can be fabricated separately and later combined into a single stack using vertical conductive posts. The vertical posts allow the 3D coprocessor to monitor and even override or disable the internal structures of a CPU, something that traditional circuit-board-level coprocessors cannot do. In this thesis, we propose a novel coprocessor architecture in which one layer houses application-specific coprocessors for cryptography and compression, which provide acceleration for applications running on a general-purpose processor, or CPU, in another layer. A compelling application for such a system is one that performs real-time dynamic program analysis of the internal structures of the CPU layer, collecting data on instructions executed, memory accesses, etc., and compressing this application trace data to a smaller size so that it can be transmitted off-chip to permanent storage for offline analysis. Furthermore, an optional encryption step, performed by the cryptographic circuitry in the coprocessor layer, can protect this compressed data from interception.

In this thesis, we set out to answer the following research question: What is a secure and cost-effective 3D architecture for the real-time transformation (compression or encryption) of a stream of data, and what are its performance characteristics? Corollary questions to be answered in pursuit of this question are: What are the requirements of a data transformation coprocessor in a three-dimensional integrated circuit? Which existing trace-file compression algorithms and architectures are optimal for compressing dynamic program-analysis data? What performance characteristics can be expected of a 3D encryptor or compressor? What are the benefits of a 3D implementation over a 2D implementation? What is the interface between the coprocessor layer and the processor layer? How is the coprocessor initialized and invoked? How do the processor and the coprocessor communicate? What modifications to the computational plane are required to support an optional control plane? What signals should be monitored, and can the interface be designed to support a wide variety of monitoring tasks? In addition to the data transformation circuitry, what other elements are required in the control plane (e.g., memory buffers, I/O controllers, etc.)?

3D integration is an emerging chip fabrication technique in which multiple integrated circuit dies are joined using conductive posts. 3D integration offers several performance and security advantages, including extremely high bandwidth between the two dies and the ability to augment a processor with a separate die housing custom security features. This thesis will perform a feasibility and requirements analysis of a data transformation coprocessor in a three-dimensional integrated circuit (3DIC). This thesis will explore the design of a 3D system consisting of one die housing a general-purpose processor and another housing a data-transformation coprocessor featuring either cryptographic or compression functions.

Traditional 2D cryptographic coprocessors are connected to general-purpose processors at the circuit-board level, or, in a multi-core, system-on-chip (SoC) at the chip level. Some processors include cryptographic functions in the instruction-set architectures (ISA). For some applications, a 2D implementation is sufficient; however, other applications may require the high bandwidth possible only with a 3D implementation.

With respect to compression, this thesis will study the principles, performance, and compression ratios of algorithms developed specifically for compressing trace files of a processor's execution and study pervasive industrial scenarios in which compression coprocessors are employed. The results will facilitate the design of a layer in a 3DIC for compressing data streams such as dynamic program profiles collected from another layer, thus enabling offline security analysis by reducing off-chip communication and storage costs.

With respect to cryptography, this thesis will study basic principles and describe various cryptographic algorithms. By studying the operation of a specific 2D cryptographic coprocessor, we will try to implement this specific coprocessor in a 3DIC design, utilized in order to encrypt the compressed data streams.

This thesis will consider how to determine the optimal 3D architecture to meet the requirements of a given application scenario that requires the transformation of data. A key architectural consideration for the 3D system is the interface between the processor and the coprocessors, how the coprocessor is invoked, and how to provide independent I/O and power to the coprocessor.

Vasudevan et al. have developed the XTRec primitive for recording the instruction-level execution trace of a commodity computing system while simultaneously ensuring the integrity of the recorded information on commodity platforms without requiring software modifications or specialized hardware [5]. Such a primitive can be used to perform postmortem analysis for forensic purposes. Our work differs from XTRec in that we are proposing a specialized 3DIC approach, and we argue that our proposed system would facilitate the capture of additional activity besides the instruction trace at higher bandwidth.

Our objective is to answer the research question by designing a system that can keep track of processor executions in real time, on an untrusted device, and send those execution traces to a trusted device for behavior analysis or to storage for subsequent forensic analysis. To do this, we propose a three-dimensional, integrated-circuit architecture, a trusted piece of hardware comprising a compression and/or cryptographic coprocessor (control plane) on top of an untrusted integrated circuit (computation plane) to collect execution traces as they are generated in the processor, then to compress and encrypt them for transmission over a common network environment. Finally, the traces are analyzed or stored on this trusted device. The proposed architecture is described in Figure 1.

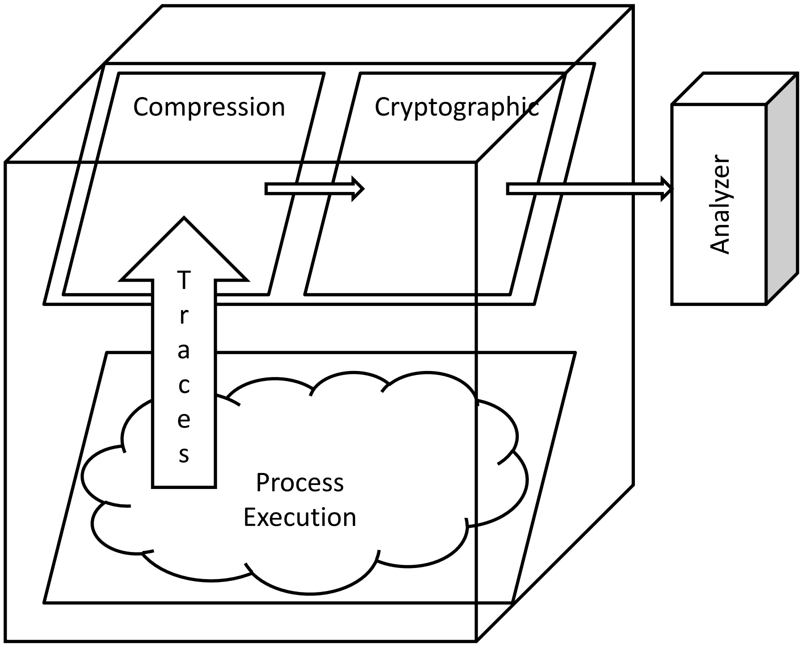


Figure 1. Block diagram of proposed system, showing the traces of a processor's execution in the CPU layer being sent to a compression device in a different layer, then being encrypted for transferral to an external analysis device. The only trusted devices are the coprocessor layer and the analyzer.

We argue that this architecture has the advantage of being faster than other architectures such as traditional coprocessors packaged separately and connected at the circuit board level, which we refer to as *off-chip* devices, or traditional 2D chips that combine a CPU and a coprocessor on the same die, which we refer to as *on-chip* devices. 3D integration offers the potential of less latency than either off-chip or on-chip because of the reduced wire length made possible by stacking. While, in on-chip devices, the collected traces have to travel a greater distance to be compressed and encrypted than in a 3DIC over internal connection in off-chip devices, the delays are even bigger, because of the distances and slower off-chip buses involved (with limited pins and inherently slow bus characteristics). In our proposed architecture, traces are collected dynamically from the CPU layer and routed vertically to a corresponding location in the coprocessor layer by means of through-silicon vias (TSV), reducing wire length and corresponding latency. A comparison of the three approaches (off-chip, on-chip, and 3DIC) is show in Figure 2.

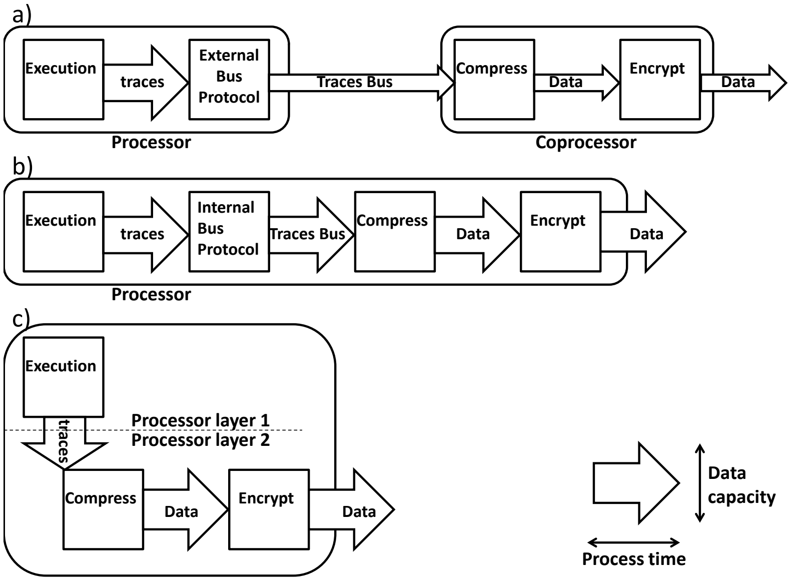


Figure 2. Comparison of three design paradigms. The horizontal axis represents time. Arrow width represents data-transfer capacity: a) coprocessor architecture ("off-chip;" bus capacity together with wire length and corresponding latency are the bottlenecks of this architecture), b) on-chip architecture (although the data capacity is high, the internal bus delay reduces the speed of the system), and c) 3DIC proposed architecture. For the latter, data that is written to the processor registers and sent via direct links to the compression registers. The collection of trace data can be turned to operate in parallel with their execution. This implementation eliminates latency associated with operation of the bus. A 3D architecture also provides shorter wire length and reduces the corresponding latency due to spatial locality enabled by stacking.

The 3DIC architecture is significant because speed is very important in the execution trace scenario. This advantage can be applied to dynamic program analysis for reverse engineering of malicious software and post-mortem analysis of a system that has suffered an attack. The amount of data collected depends on the granularity of the collection and the speed of the system. Monitoring and collecting more signals results in a larger data stream. The compression of data before it is transferred increases the bandwidth to off-chip storage.

Different compression scenarios call for different solutions. For example, in common image-compression applications, incoming information can be a periodically delayed or deleted (e.g. when input buffers are full) without significantly compromising the fidelity of the reconstructed image. However, when collecting execution traces we do not want to drop any data or slow down the whole system. Loss of important trace data can be disastrous.

We argue that, although a 3DIC can cost more, our approach offers several key benefits. For example, the CPU layer can be sold to ordinary customers without the coprocessor layer attached, but customers with high trustworthiness requirements can purchase the joined unit [4]. Moreover, the CPU layer can be manufactured in an untrusted foundry, while the coprocessor layer is manufactured in a trusted foundry to provide requisite trustworthiness to the combined system. This approach could improve the economic feasibility of trustworthy-system acquisition.

Trustworthy 3D systems can be used for purposes such as protecting information in government, e-business, banking, and voting machines and providing highly trustworthy audit and program analysis in real time.

A 3DIC architecture also enables stacking different technologies and architectures, each optimized separately for its unique purpose [6].

A 3DIC implementation of a data-transformation coprocessor has the potential to significantly reduce the cost of collecting large amounts of dynamic program-analysis data for offline security analysis, in terms of transmission and storage cost. For example, the results of this thesis will be useful in a future implementation of a system that uses a 3DIC to reduce the cost of reverse-engineering malicious software and non-security program profiling. The results of this work can also be used to enhance systems with custom security functions in a cost-effective and computationally efficient manner. For example, systems that would traditionally employ a separate crypto coprocessor chip will be able to use a 3D crypto coprocessor, providing performance benefits for high-throughput applications as well as security benefits. For example, the crypto coprocessor can be fabricated in a trusted foundry, and both crypto transformations and key storage can be decoupled from the computation plane).

## B. SCOPE OF THESIS

This thesis will explore the architecture and design of 3DIC systems consisting of a general-purpose processor die (computational plane) joined with another die (control plane), housing either a crypto or compression coprocessor. This will require an understanding of 3DIC integrated circuit implementation, coprocessors implementation, and existing compression and encryption schemes.

This thesis will also present, for comparison purposes, the 2D implementation of a system consisting of a general-purpose processor with a crypto or compression coprocessor, and the 3DIC implementation of such a system.

For each design parameter for the strawman design (e.g., the interface between the two dies, the method of communication between them, the method of configuring the control plane, the elements residing in the control plane, the fabrication process, the modifications to the computation plane, etc.), a choice will be made. The argument for each choice will be defended based on analysis of real 3D systems described in the literature. An analysis of the performance of the 3D implementation will be made based on published figures from the literature, an estimate of the number and size of the vertical connections, and traditional 2D architectural simulation. In addition, binary instrumentation [7] will be used to generate a variety of trace files representing dynamic program-analysis data from the computation plane. These trace files will be used to compare the performance and compression ratio for a variety of existing trace compression algorithms. For those compression algorithms that are normally implemented in hardware, a software program emulating the behavior of the hardware will be used.

## C. THESIS OUTLINE

This thesis is organized into the following chapters. Chapter II describes the concepts, terminology, and commonalities of compression and cryptographic transformations and provides a background on 3DIC technology, coprocessors, data compression, and cryptography.

Chapter III covers compression algorithms, 2D compression coprocessors, usage scenarios, and performance numbers. Chapter IV covers cryptographic algorithms, 2D cryptographic coprocessors, usage scenarios, and performance numbers.

Chapter V compares and contrasts various factors in 3D architectures, and identifies options for a strawman design for real-time transformation of computation plane data~~.~~ Chapter VI describes the ideal 3D system, analyzes the requirements for the interface between the processor and coprocessor layers, describes the elements required in the control plane, as well as configurable parameters of the crypto and compression circuitry, to support a variety of crypto and compression tasks. It also develops estimates of the ideal system's performance in compression and cryptography. Chapter VII summarizes the analytical results and describes open issues that call for future work.

# II. BACKGROUND

## A. INTRODUCTION

A large variety of modern technologies such as computer networks and electronic commerce, demand private and secure communications for everyday transactions. Compression and cryptography provide a useful set of primitives, methods, and modes of operation to support fast, accurate, reliable, and secure data transmission.

David Solomon defines transformation as “a mathematical operation that changes the representation of a data item. Thus, changing the decimal number 12,345 to the binary 11000000111001 is a transform” [8].

In the same way we can think about data transformation in the field of compression: we can transform audio data, or image pixels to a representation that requires fewer bits. David Solomon describes two simple transformation techniques for compression: “The transformed items (transform coefficients) are decorrelated. Such a transform already achieves some degree of compression, but more can be obtained if lossy compression is an option. The transform coefficients can be quantized, a process that results in small integers (which can be encoded with variable-length codes) and possibly also in runs of zeros (which can be compressed with RLE)” [8]. Similarly, in the field of cryptography, ciphers transform plaintext to ciphertext to protect information from unauthorized access. Ciphers, together with cryptographic hash functions, help protect the confidentiality and integrity of data.

We argue that incorporating compression and cryptographic functionality into a coprocessor embedded in the control plane of a 3DIC can improve the performance of data transformation significantly.

## B. 3D IC TECHNOLOGY

3D integration is an emerging technology in which two or more integrated circuit (IC) die are fabricated separately and later bonded together into a single stack. The layers are connected using conductive posts to form a single circuit, and the entire stack is contained in a single package. Advantages of 3D integration include lower power, high bandwidth between dies, reduced latency, the ability to join disparate technologies, and the ability to control the lineage of a subset of the dies, e.g., by manufacturing them in a trusted foundry.

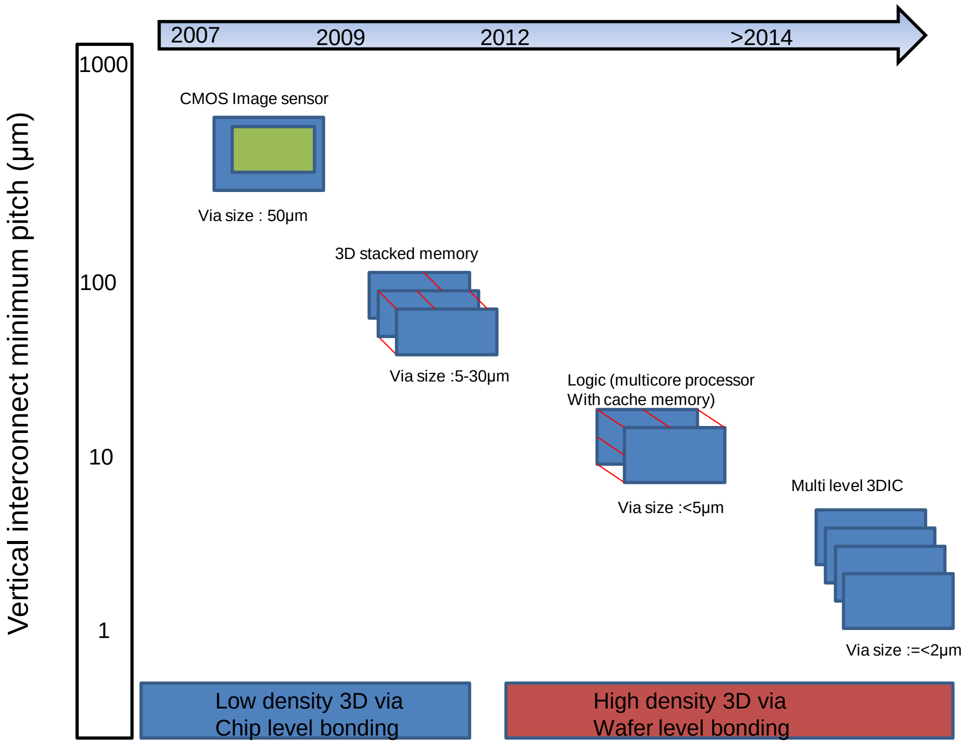


Figure 3. Application Trend of 3D Silicon Integration (After [10]). We note the rapid increase of 3D technology within a small period of time (approximately two years) in conjunction with a significant reduction of the interconnect-via size, from 50µm to less than 2µm. This is important because reducing via size also reduces wire length and, consequently, the thermal effects of a 3DIC.

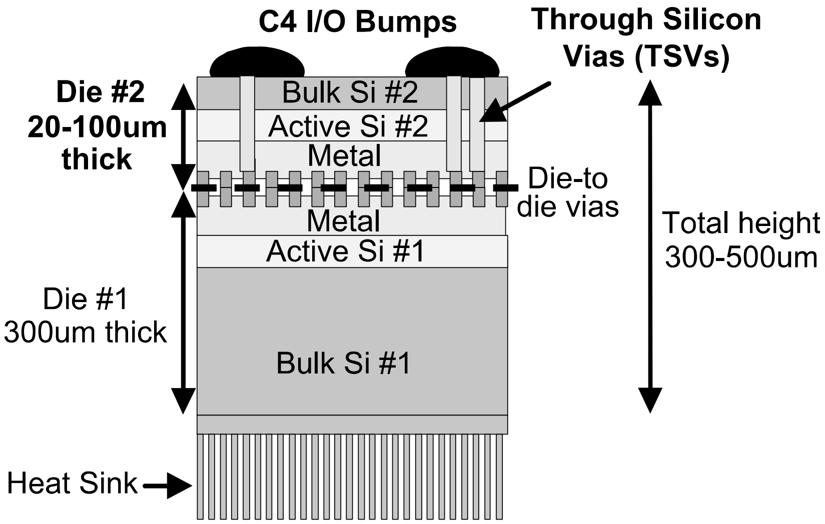


Figure 4. A three dimensional (3D) structure (From [17]). Two dies are joined together using face-to-face bonding. The die-to-die vias connect the two dies together. Through-silicon vias (TSVs) serve two purposes: (1) they provide interconnection between the C4 I/O bumps and the active region of die; and (2) they satisfy power-delivery requirements. A heat sink is used for the dissipation of the heat.

### 1. Main Technologies for Manufacturing 3DICs

Various 3D technologies are being explored in industry and academia [11], but the two most used and promising are wafer-bonding [6], [12] and multi-layer, buried structures (MLBS) [13]. Wafer-bonding technology fabricates each active device layer separately on a different wafer and then bonds the wafers to form a single entity. On the other hand, with MLBS, multiple active device layers are fabricated on a single wafer before a back-end process builds interconnections between the devices. In general, there are two basic ways of stacking dies: *face-to-face* and *face-to-back*, where the ‘‘face’’ refers to the side that supports the metallization (i.e. logical circuits), and the ‘‘back’’ refers to the side with the silicon substrate [6]. Both of these stacking methods will be analyzed in depth in Chapter V, where we will study various 3DIC architectural considerations. For face-to-face bonding, a copper–copper bonding process is used to construct interdie connections, also called die-to-die (d2d) or face-to-face vias. The process achieves face-to-face vias by depositing the copper material of half of the via on each die, and then bonding the two dies together, utilizing a thermocompression process.

Finally, a chemical–mechanical polishing (CMP) process thins the back of one die to reduce the thickness of the bulk silicon. Face-to-face vias are smaller than the through-silicon vias required for face-to-back bonding [14]. On the other hand, for 3DICs composed of more than two active layers, face-to-back bonding is the only option.

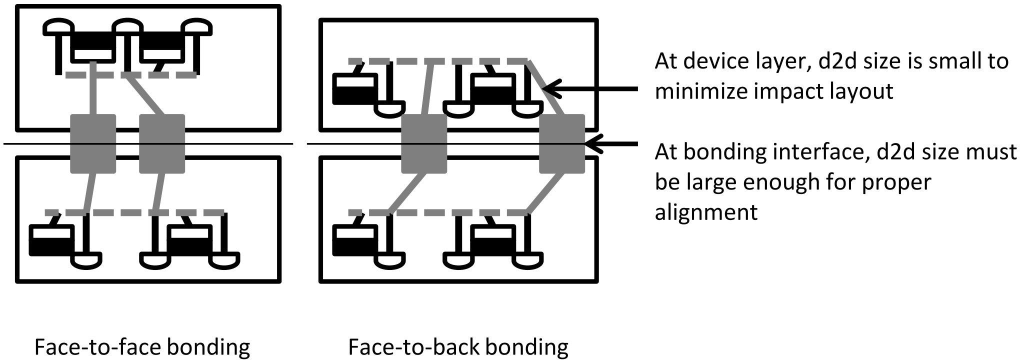

At device layer, d2d size is small to minimize impact layout
At bonding interface, d2d size must be large enough for proper alignment
Face-to-face bonding
Face-to-back bonding

Figure 5. Bonding orientation (From [6]). This figure shows face-to-face and face-to-back bonding. Depending on the position of the metal layers of the upper die relative to those of the lower die, the bonding process is referred to as either *face-to-face*, where the metal layers of the layers face each other, or *face-to-back*, where the bulk silicon of the upper die faces the metal layers of the lower die.

### 2. Advantages of 3DIC Technology

3DIC technology offers several potential advantages. Industry is pursuing 3D integration to increase the number of transistors on a chip as an alternative to the costly retooling required to make transistors that are smaller than 22nm [14]. Moreover, much like tall skyscrapers allow more activity within the same footprint, 3DIC technology increases the number of transistors that can be placed on the same footprint by adding transistors vertically.

Another advantage of 3D integration is flexibility and modularity, because a high-performance processor die can be optionally joined with application-specific dies that perform custom functions such as acceleration.

Another advantage of 3DIC technology is that it facilitates the combination of dissimilar technology, e.g., dies fabricated separately using different processes that are optimized for the needs of the individual die and later joined into a single, unified stack. This has the potential to reduce costs by reusing dies that have been designed, tested, and certified, provided that the necessary modifications to support 3D stacking have been made. Another benefit of 3D integration is that we can optionally join a die with a die that implements custom security functions, enhancing the security of a system that requires them. For instance, a crypto coprocessor can be manufactured in a separate layer called the control plane, where all crypto operations will execute. This control plane is separate from the computational plane that utilizes the crypto coprocessor.

We summarize the benefits of 3DIC technology as follows [11]: 1) it offers higher transistor density per footprint over conventional 2D layout and an increase in the number of transistors that can be used; 2) The reduction of the total wire length required for the interconnections leads to an enhancement of the performance; and 3) it has lower power requirements [16]. It has been shown [16] that the reduction of wire length resulting from the use of three-dimensional architectures is proportional to a factor of the square root of the number of layers used. Assuming that we have a four-layer 3DIC, we can achieve on average $\sqrt{4} = 2$ times shorter wiring length as depicted in Figure 6.

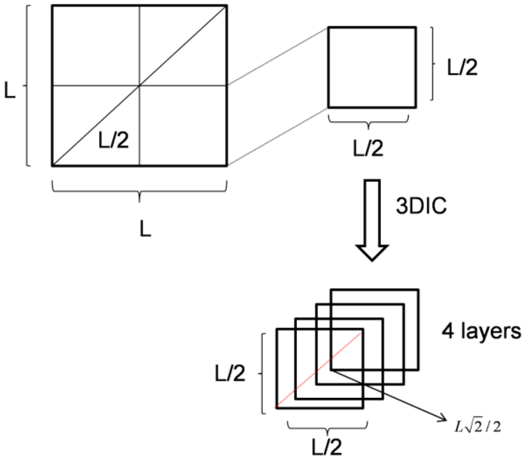


Figure 6. Reduction of length wiring by a factor of the square root of the number of layers in three dimensions (After [11]). For a 3DIC with four layers, the average reduction of the length wiring is a factor of two.

### 3. Challenges of 3D IC technology

3DIC technology has several clear drawbacks, just as skyscrapers do [15]. 1) Thermal effects and the need to cool a stack of dies present a challenge. New cooling methods are needed to prevent or eliminate thermal problems. 2) Yield: Each additional manufacturing step adds a risk for potential defects. In order for 3DICs to be reliable and commercially viable, these defects must be avoided or repaired, and 3DICs must be tested and certified to operate properly. 3) New testing methods should be implemented. The implementation of the requisite test tools and procedures is occurring in parallel with the implementation of 3DIC technology. Current testing methods and tools for conventional 2DICs are not compatible with 3DIC technology. This impacts research, manufacturing time, certification, and fabrication cost. 4) Heterogeneous integration supply chain: In heterogeneously integrated systems, the delay of one part from one 3DIC manufacturer delays the delivery of the whole product, because the manufacturing process depends on all participating parties.

## C. COPROCESSORS

Coprocessors are application-specific integrated circuits that complement and accelerate a main processor, typically a general-purpose CPU. CPUs and their coprocessors can be connected in a variety of ways, from separate chips connected at the circuit-board level to separate cores on the same chip connected by on-chip buses, networks, or direct connections. Coprocessors can be upgraded without replacing the whole system (unless, of course, they reside on the same die). Furthermore, since they are application-specific, they have the potential of having much higher throughput and much lower power consumption than a general-purpose processor for certain application workloads that can benefit from the acceleration they offer.

When designing coprocessors, several aspects have to be considered:

- Onto what kind of slot or interface the device will be attached
- How many pins the device and slot have; this will impose limitations on the speed of the communication
- How the clock signal will be delivered to the device and whether the device will operate in sync with the main processor

- How the device will communicate with the main processor and other devices (microprocessor interface)

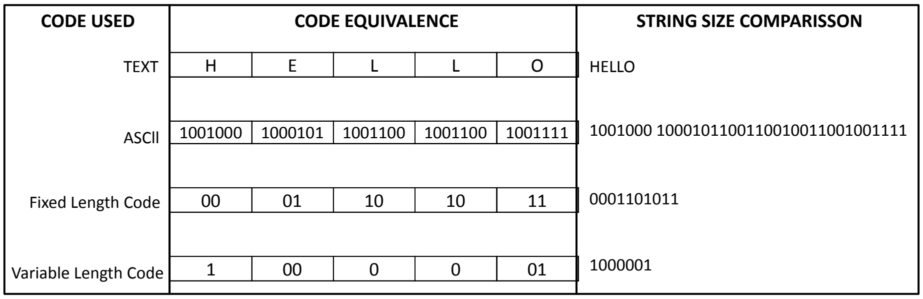
What are the input/output signals (are buffers needed?)

## D. DATA COMPRESSION

### 1. Compression

Compression is basically a redundancy elimination method, so the first step is to find the redundancy and its cause. David Salomon in [8] presents some simple explanations of data compression, as discussed below.

The simple compression example is the use of variable-length codes to represent symbols. As an example, characters can be represented by ASCII or Unicode. Both are fixed-length codes, but we know that in English the most common letters are E, T, and A, while J, Q, and Z are least common. Therefore, a basic compression method is just to choose the characters that appear more frequently and represent them with fewer bits, instead of representing all characters with the same number of bits, as presented in Figure 7.



| CODE USED | CODE EQUIVALENCE | | | | | STRING SIZE COMPARISSON |
|---|---|---|---|---|---|---|
| TEXT | H | E | L | L | O | HELLO |
| ASCII | 1001000 | 1000101 | 1001100 | 1001100 | 1001111 | 1001000 10001011001100100110010011111 |
| Fixed Length Code | 00 | 01 | 10 | 10 | 11 | 0001101011 |
| Variable Length Code | 1 | 00 | 0 | 0 | 01 | 1000001 |

Figure 7. Here we present the difference in length of using a fixed-length code versus a variable-length code to represent a simple text message. In the first row, we have a text message as it appears to the user; in the second row, the way it appears to the computer if using ASCII fixed-length representation; in the third row we represent it using an arbitrary fixed-length code specific to a four-character alphabet, to reduce the string length; and in the last row, we replace the character that appears most (L in this case) by the fewest possible number of bits (0 in this case). Then we look for the second most-used character. Since all the remaining characters appear only once, we just assign them a different code.

The same approach can be used to compress images, which are made of pixels. To represent all recognizable colors, we need a large range of possible color values for each pixel, so we can use 24 bits to represent one pixel. On the other hand, black and white or grayscale pixels can be represented using a smaller variable-length code and assigning fewer bits to the most-used colors. This reduces the overall size of the image file. This can be seen in Figure 8.

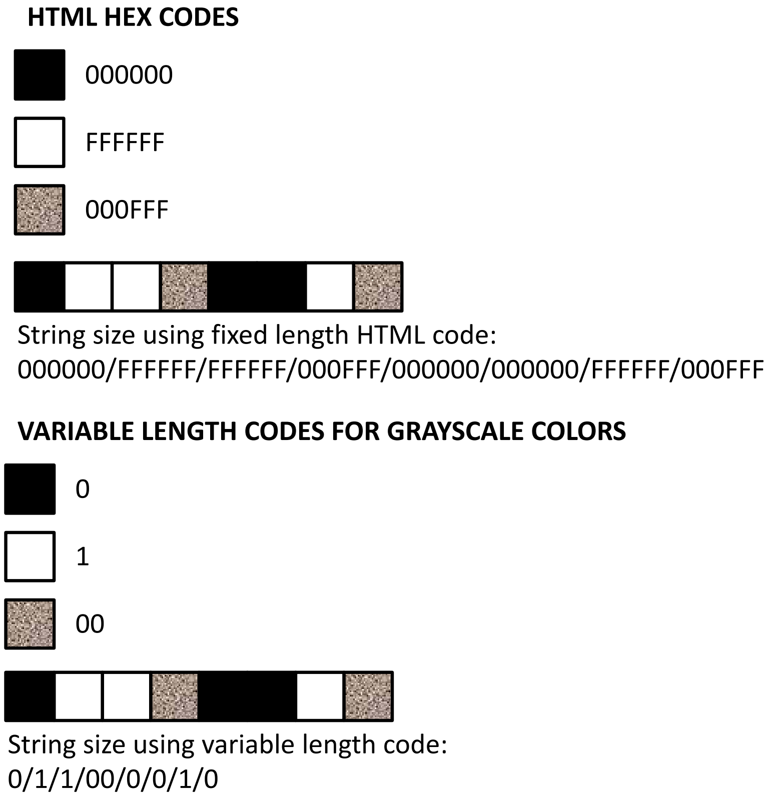

HTML HEX CODES
000000
FFFFFF
000FFF
String size using fixed length HTML code:
000000/FFFFFF/FFFFFF/000FFF/000000/000000/FFFFFF/000FFF
VARIABLE LENGTH CODES FOR GRAYSCALE COLORS
0
1
00
String size using variable length code:
0/1/1/00/0/0/1/0

Figure 8. We present the difference in length using a fixed-length code or variable-length code to represent a grayscale image.

The drawback of this method is that the system has to be able to recognize each character or pixel representation, which is called uniquely decodable or uniquely decipherable (UD), as explained by David Salomon in [8]: “Once the original data symbols are replaced with variable-length codes, the result (the compressed file) is a long string of bits with no separators between the codes of consecutive symbols. The decoder (decompressor) should be able to read this string and break it up unambiguously into individual codes. We say that such codes have to be uniquely decodable or uniquely decipherable (UD).” This separation can be done using a time delay between characters in the same symbol, and a different time delay between different symbols, or using a special character to indicate the end of one symbol, any time the code increases in complexity or size.

Now consider the previous image example, and suppose we have an image from an orange. It is likely that we will find a large sequence of orange pixels, so we can replace this sequence with just the first pixel followed by the number of times this pixel appears. This method is called run-length encoding and can be seen in Figure 9.

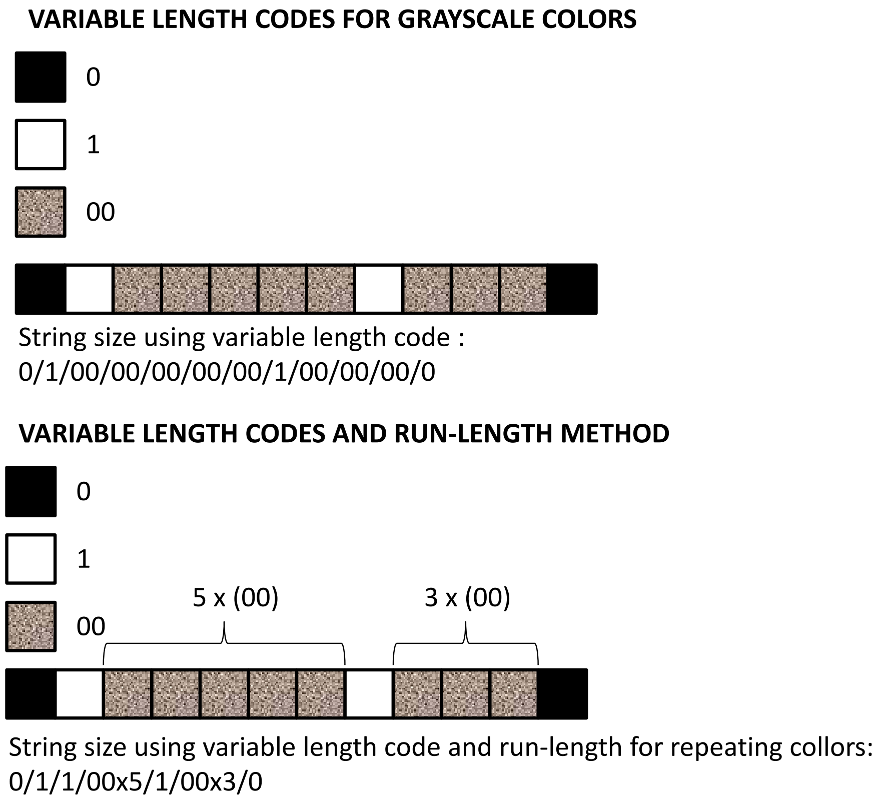

VARIABLE LENGTH CODES FOR GRAYSCALE COLORS
0
1
00
String size using variable length code :
0/1/00/00/00/00/00/1/00/00/00/0
VARIABLE LENGTH CODES AND RUN-LENGTH METHOD
0
1
00
5 x (00)
3 x (00)
String size using variable length code and run-length for repeating collors:
0/1/1/00x5/1/00x3/0

Figure 9. Here we present the difference in the string length by using a run-length method to represent an image with repeated colors.

## 2. Dictionary methods

Dictionary compression methods rely on the principles described above. Dictionary compression, as its name suggests, inputs data and stores it in a special structure called a dictionary, outputting a pointer to its location (token). The pointer is smaller than the data stored to that location, so compression is achieved by substituting the original data with its pointer. David Salomon and Giovanni Motta in [9] describe dictionary methods as methods that "(…) do not use a statistical model, nor do they use variable length codes. Instead they select strings of symbols and encode each string as a *token* using a dictionary. The dictionary holds strings of symbols, and it may be static or dynamic (adaptive). The former is permanent, sometimes permitting the addition of strings but no deletions, whereas the latter holds strings previously found in the input stream, allowing for additions and deletions of strings as new input is being read." During the whole compression process, the algorithm performs a search in the dictionary, looking for redundancy (a repeating string that was previously seen and stored in the dictionary) and outputs a pointer to the longest match (the string with the greatest number of equal characters). If no match is found, this string is stored as a new entry in the dictionary, and a pointer to its location is the output. Figure 10 shows a simple dictionary compression example and Figure 11 shows the decompression of the same string. Various dictionary-based methods have been developed; the major difference between them is how they handle the process of storing to and searching the dictionary.

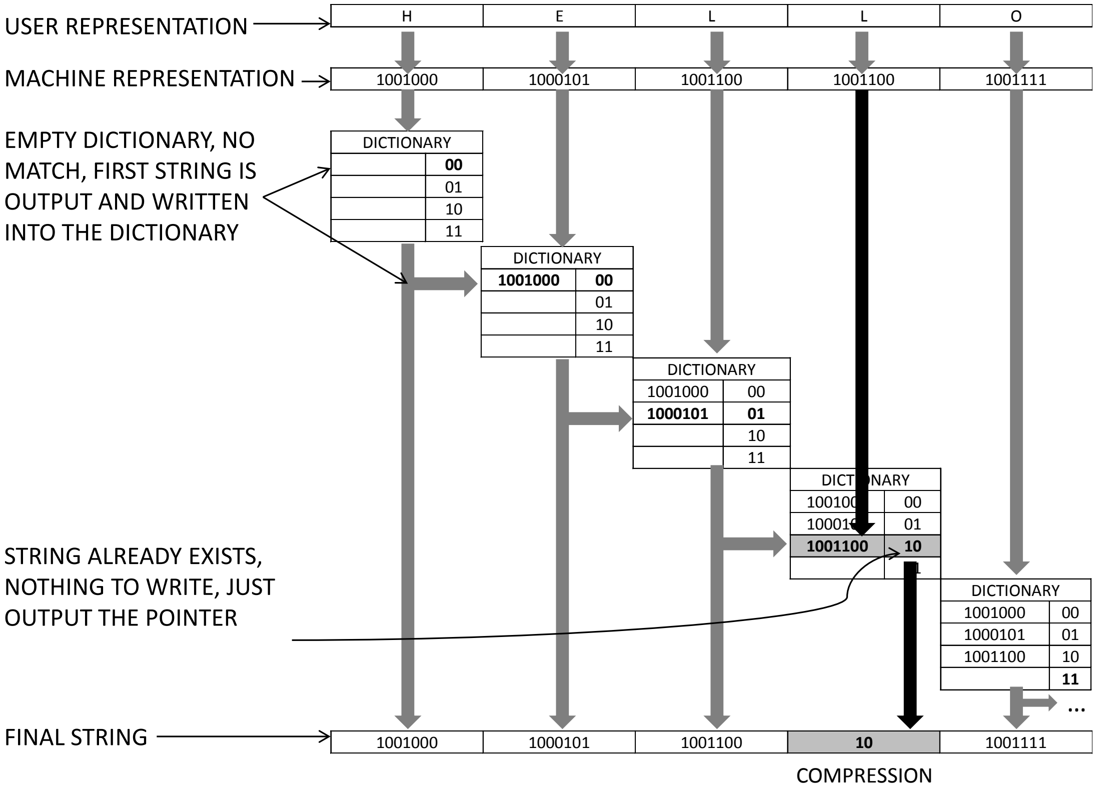


Figure 10. We present the dictionary method with just one match. At first it seems that no compression can be achieved, but as the dictionary fills up, more matches occur. The methods differ in the way they handle the filling process and how they manage the dictionary when it is full and more new data is found.

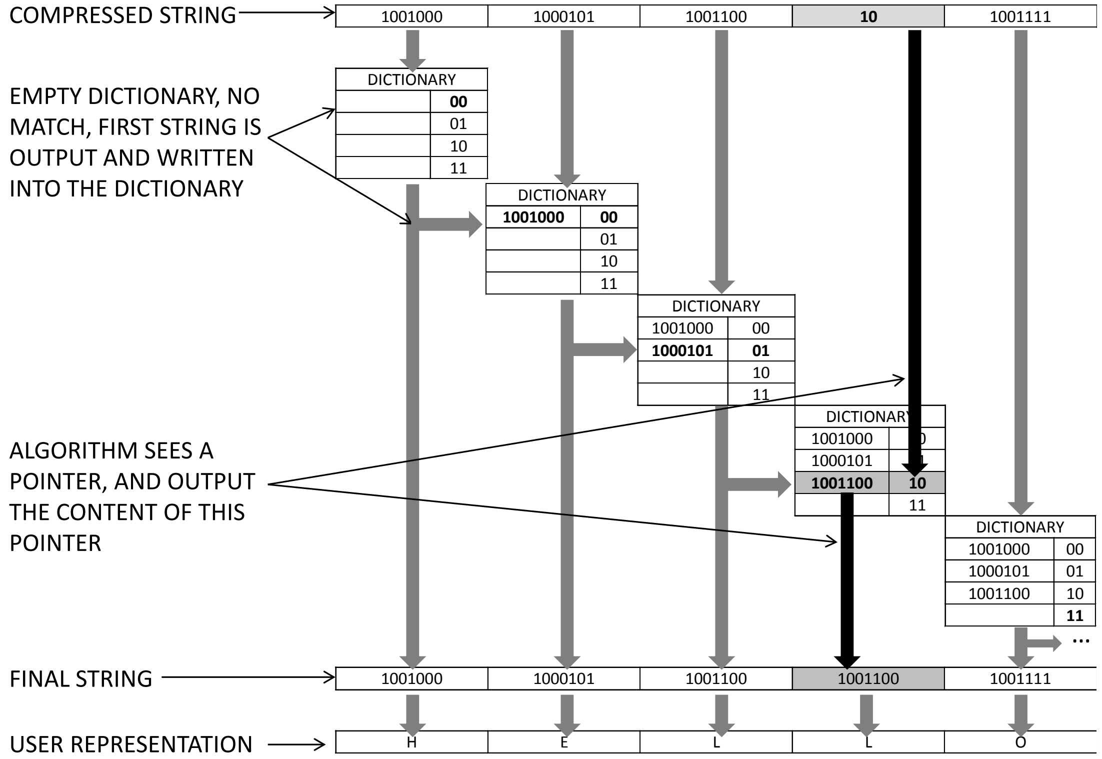


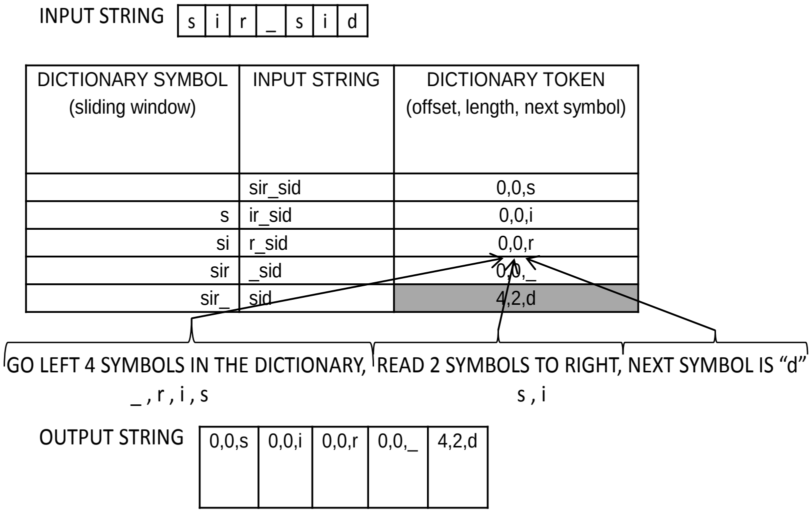
Figure 11. Using the compressed string from Figure 10, we present the decompression phase. We assume that the algorithm can figure out if the input data is compressed or not. If it's not compressed, the data is output and written to the dictionary as is; otherwise the algorithm interprets it as a pointer and outputs the pointer's content.

### *a. Adaptive Dictionary*

We can specify each dictionary method and show the difference between them. In [9] David Salomon and Giovanni Motta show that "in general, an adaptive dictionary-based method is preferable. Such a method can start with an empty dictionary or with a small, default dictionary, add words to it as they are found in the input stream, and delete old words because a big dictionary slows down the search".

### *b. Sliding Window Dictionary*

Some methods use a sliding window over the previously seen inputs as a dictionary, and the token (output of compressed data) is the triple (offset from the actual symbol into the sliding window, from right to left, the length of the repeating symbols from left to right, and next-input symbol). Figure 12 shows an example.



INPUT STRING

| s | i | r | _ | s | i | d |
|---|---|---|---|---|---|---|

| DICTIONARY SYMBOL (sliding window) | INPUT STRING | DICTIONARY TOKEN (offset, length, next symbol) |
|---|---|---|
| | sir_sid | 0,0,s |
| s | ir_sid | 0,0,i |
| si | r_sid | 0,0,r |
| sir | _sid | 0,0,_ |
| sir_ | sid | 4,2,d |

OUTPUT STRING

| 0,0,s | 0,0,i | 0,0,r | 0,0,_ | 4,2,d |
|---|---|---|---|---|

Figure 12. We present a different implementation of a dictionary in a sliding window; this shows how algorithms vary based on the implementation of the same principles (After [9]). As an example of how to consider the method based on the kind of data we are compressing, this method compares the input string only to the sliding window, not the full data previously seen, so we assume our data will repeat sooner.

### *c. Circular Queue Dictionary*

A circular queue is another implementation of the dictionary. Instead of having a sliding window, which has to shift its entire content on each input data, the circular queue is just a circular array, a linear array physically, that inputs data sequentially, but when the data reaches the last available position in the string, the pointer is redirected to the first position and data is overwritten.

#### *d. Binary Tree Dictionary*

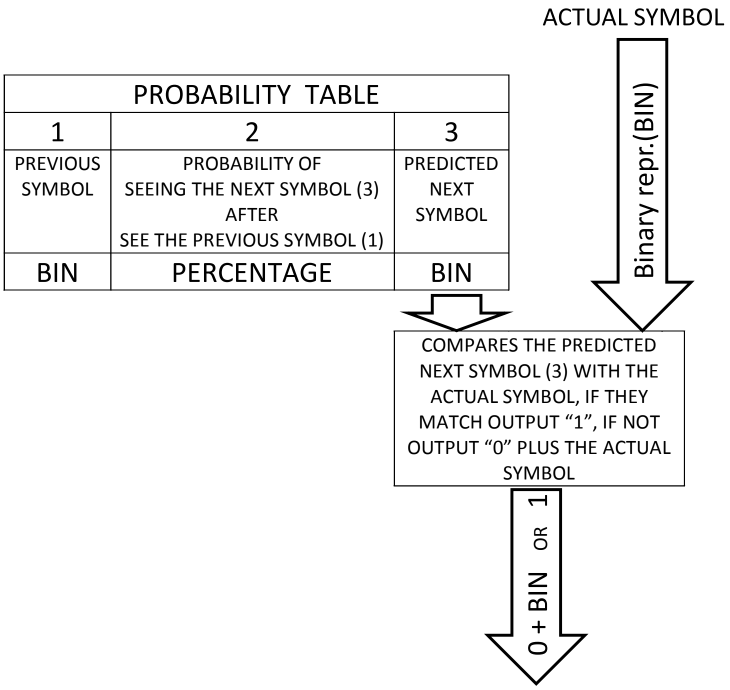
Another implementation of dictionaries uses a binary search tree sorted in lexicographical order. A binary search tree is defined in [9] as "… a binary tree where the left sub tree of every node *A* contains nodes smaller than *A*, and the right sub tree contains nodes greater than *A*," where "smaller" means that the string appears first in the dictionary.

The reason for this implementation is that the search process in a balanced binary search tree is faster, because the number of steps needed to find a node is the height of the tree, in this case $\log_2$ n (where *n* is the number of elements in the tree).

### 3. Statistical Methods

Prediction methods take advantage of the context in which data appears. After inputting a certain amount of data, the method (predictor) is able to guess (predict) the next input based on the context in which it appears, by means of statistical analysis. That is why the prediction method is a subset of all statistical methods. For example, we know from the English alphabet that TH is the most common digram, so after seeing a T we expect an H with high probability. In the same way, the predictor, after inputting a large amount of data, generates its own statistics, assign probabilities for the next symbol, based on context, and chooses the most probable one to output. Good predictors are able to correctly guess more than 90% of the output data.

Prediction-based methods have a learning or heating phase in order to generate necessary statistics before being able to make good predictions. In figures 13 and 14, we show this learning process step-by-step.



| PROBABILITY TABLE | | |
|---|---|---|
| 1 | 2 | 3 |
| PREVIOUS SYMBOL | PROBABILITY OF SEEING THE NEXT SYMBOL (3) AFTER SEE THE PREVIOUS SYMBOL (1) | PREDICTED NEXT SYMBOL |
| BIN | PERCENTAGE | BIN |

ACTUAL SYMBOL
Binary repr.(BIN)
COMPARES THE PREDICTED NEXT SYMBOL (3) WITH THE ACTUAL SYMBOL, IF THEY MATCH OUTPUT “1”, IF NOT OUTPUT “0” PLUS THE ACTUAL SYMBOL
0 + BIN OR 1

Figure 13. Before presenting the overall compression process, we show the probability table (in percentage): the representation of an algorithm that, based in the previously seen symbol, having learned about the data (by copying to the table the previously seen symbol and actual following symbols and counting how many times they appear), can guess the most-probable next symbol. If the guess matches the actual symbol we have a hit, and the output is the smallest possible data (“1” in our example). If the guess doesn’t match the actual symbol, the algorithm outputs a miss symbol (“0” in our example), followed by the actual symbol.

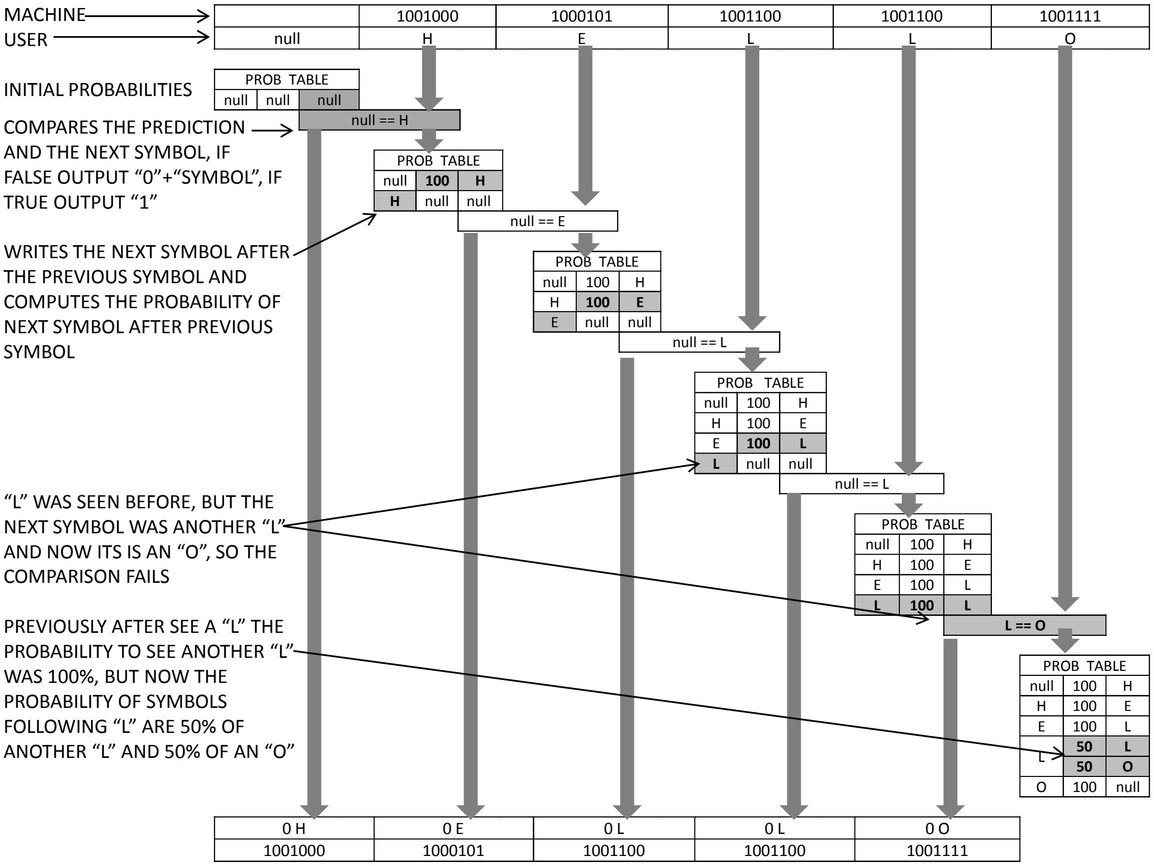
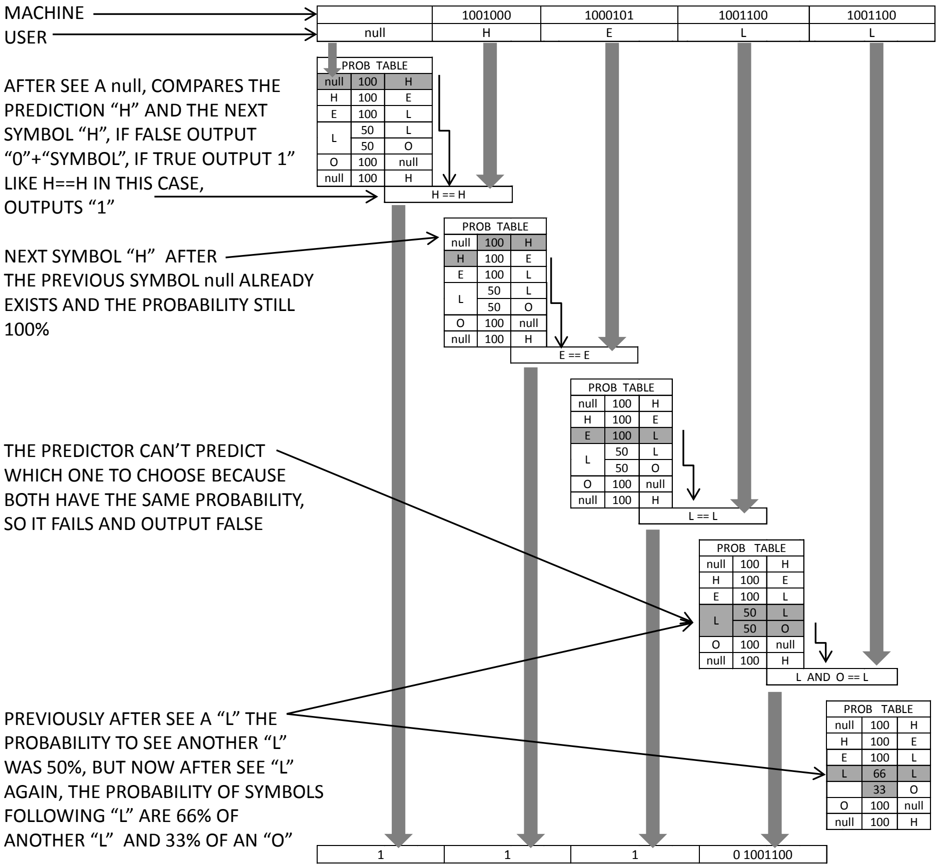

MACHINE
USER
1001000
1000101
1001100
1001100
1001111
null
H
E
L
L
O
INITIAL PROBABILITIES
PROB TABLE
null null null
COMPARES THE PREDICTION AND THE NEXT SYMBOL, IF FALSE OUTPUT "0"+"SYMBOL", IF TRUE OUTPUT "1"
null == H
PROB TABLE
null 100 H
H null null
WRITES THE NEXT SYMBOL AFTER THE PREVIOUS SYMBOL AND COMPUTES THE PROBABILITY OF NEXT SYMBOL AFTER PREVIOUS SYMBOL
null == E
PROB TABLE
null 100 H
H 100 E
E null null
null == L
PROB TABLE
null 100 H
H 100 E
E 100 L
L null null
null == L
"L" WAS SEEN BEFORE, BUT THE NEXT SYMBOL WAS ANOTHER "L" AND NOW ITS IS AN "O", SO THE COMPARISON FAILS
PROB TABLE
null 100 H
H 100 E
E 100 L
L 100 L
L == O
PREVIOUSLY AFTER SEE A "L" THE PROBABILITY TO SEE ANOTHER "L" WAS 100%, BUT NOW THE PROBABILITY OF SYMBOLS FOLLOWING "L" ARE 50% OF ANOTHER "L" AND 50% OF AN "O"
PROB TABLE
null 100 H
H 100 E
E 100 L
L 50 L
50 O
O 100 null
0 H
0 E
0 L
0 L
0 O
1001000
1000101
1001100
1001100
1001111

Figure 14. Learning phase: Unlike the dictionary method, the prediction is slow to learn about data, and, as we can see, augments the output string during this phase, adding the miss symbol "0" plus the actual data. After this learning phase, the final compression is better in terms of compression ratio than the dictionary-based method.



| | | | | | |
|---|---|---|---|---|---|
| MACHINE | | 1001000 | 1000101 | 1001100 | 1001100 |
| USER | null | H | E | L | L |

AFTER SEE A null, COMPARES THE PREDICTION “H” AND THE NEXT SYMBOL “H”, IF FALSE OUTPUT “0”+“SYMBOL”, IF TRUE OUTPUT 1” LIKE H==H IN THIS CASE, OUTPUTS “1”

NEXT SYMBOL “H” AFTER THE PREVIOUS SYMBOL null ALREADY EXISTS AND THE PROBABILITY STILL 100%

THE PREDICTOR CAN’T PREDICT WHICH ONE TO CHOOSE BECAUSE BOTH HAVE THE SAME PROBABILITY, SO IT FAILS AND OUTPUT FALSE

PREVIOUSLY AFTER SEE A “L” THE PROBABILITY TO SEE ANOTHER “L” WAS 50%, BUT NOW AFTER SEE “L” AGAIN, THE PROBABILITY OF SYMBOLS FOLLOWING “L” ARE 66% OF ANOTHER “L” AND 33% OF AN “O”

| PROB | TABLE | |
|---|---|---|
| null | 100 | H |
| H | 100 | E |
| E | 100 | L |
| L | 50 | L |
| | 50 | O |
| O | 100 | null |
| null | 100 | H |

| PROB | TABLE | |
|---|---|---|
| null | 100 | H |
| H | 100 | E |
| E | 100 | L |
| L | 50 | L |
| | 50 | O |
| O | 100 | null |
| null | 100 | H |

| PROB | TABLE | |
|---|---|---|
| null | 100 | H |
| H | 100 | E |
| E | 100 | L |
| L | 50 | L |
| | 50 | O |
| O | 100 | null |
| null | 100 | H |

| PROB | TABLE | |
|---|---|---|
| null | 100 | H |
| H | 100 | E |
| E | 100 | L |
| L | 50 | L |
| | 50 | O |
| O | 100 | null |
| null | 100 | H |

| PROB | TABLE | |
|---|---|---|
| null | 100 | H |
| H | 100 | E |
| E | 100 | L |
| L | 66 | L |
| | 33 | O |
| O | 100 | null |
| null | 100 | H |

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 1001100 |

Figure 15. After the learning phase, suppose the string continues and after “HELLO” we see “HELL;” the table is still being updated with new probabilities, but now that the algorithm has enough knowledge about our data, more hits are generated. Therefore, the compression ratio increases, and just 11 bits are output from a 28-bit input, almost a 3:1 compression ratio. Note that the symbol “L” appears for the third time after another “L” and changes the probability of symbols after “L” from 50% of ”L” and 50% of “O” to 66% of “L” and 33% of “O”.

In Figures 16 and 17 we present the decompression phase of the same string.

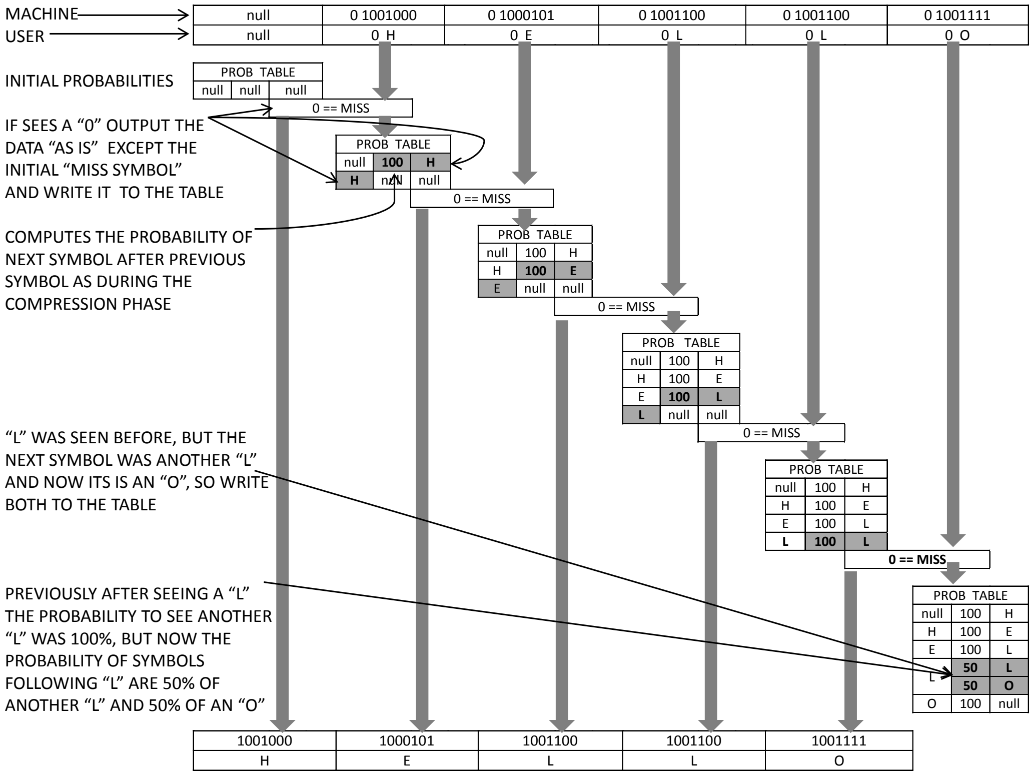

MACHINE
USER
null
0 1001000
0 1000101
0 1001100
0 1001100
0 1001111
null
0 H
0 E
0 L
0 L
0 O
INITIAL PROBABILITIES
PROB TABLE
null null null
0 == MISS
IF SEES A "0" OUTPUT THE DATA "AS IS" EXCEPT THE INITIAL "MISS SYMBOL" AND WRITE IT TO THE TABLE
PROB TABLE
null 100 H
H null null
0 == MISS
COMPUTES THE PROBABILITY OF NEXT SYMBOL AFTER PREVIOUS SYMBOL AS DURING THE COMPRESSION PHASE
PROB TABLE
null 100 H
H 100 E
E null null
0 == MISS
PROB TABLE
null 100 H
H 100 E
E 100 L
L null null
0 == MISS
"L" WAS SEEN BEFORE, BUT THE NEXT SYMBOL WAS ANOTHER "L" AND NOW ITS IS AN "O", SO WRITE BOTH TO THE TABLE
PROB TABLE
null 100 H
H 100 E
E 100 L
L 100 L
0 == MISS
PREVIOUSLY AFTER SEEING A "L" THE PROBABILITY TO SEE ANOTHER "L" WAS 100%, BUT NOW THE PROBABILITY OF SYMBOLS FOLLOWING "L" ARE 50% OF ANOTHER "L" AND 50% OF AN "O"
PROB TABLE
null 100 H
H 100 E
E 100 L
L 50 L
50 O
O 100 null
1001000
1000101
1001100
1001100
1001111
H
E
L
L
O

Figure 16. The probability table is empty, and a new learning phase will start for decompressing the string. Every time the algorithm sees a "0," it will output the data that follows and write the data into the table, calculating the appropriate probability.

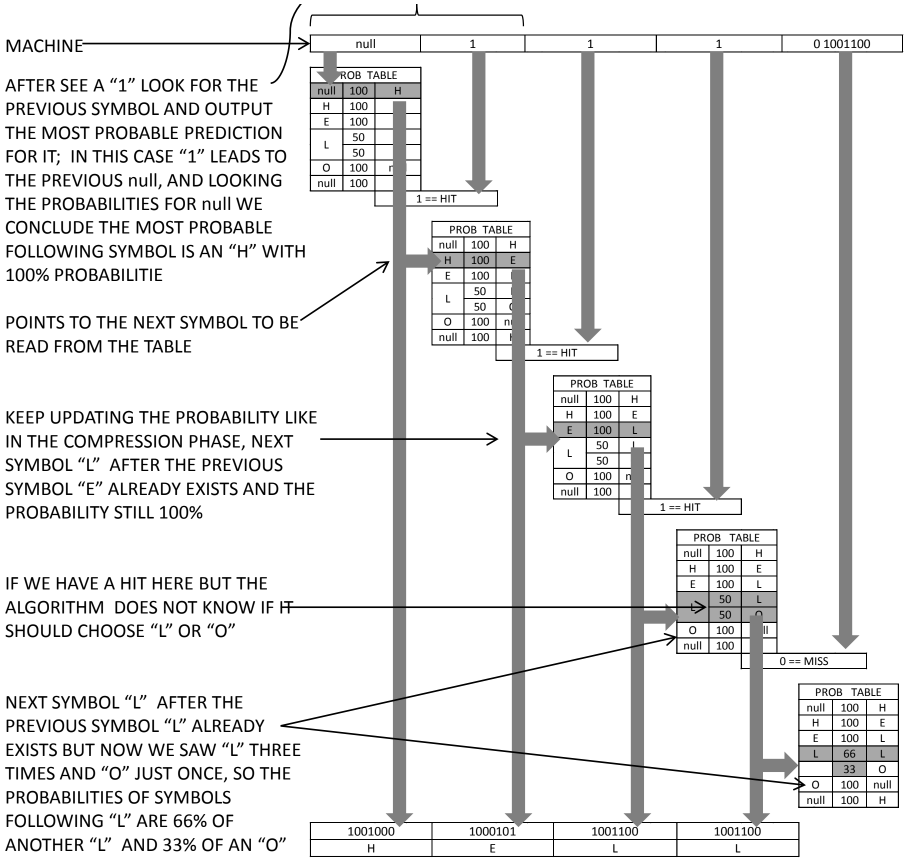

MACHINE
null
1
1
1
0 1001100
AFTER SEE A “1” LOOK FOR THE PREVIOUS SYMBOL AND OUTPUT THE MOST PROBABLE PREDICTION FOR IT; IN THIS CASE “1” LEADS TO THE PREVIOUS null, AND LOOKING THE PROBABILITIES FOR null WE CONCLUDE THE MOST PROBABLE FOLLOWING SYMBOL IS AN “H” WITH 100% PROBABILITIE
POINTS TO THE NEXT SYMBOL TO BE READ FROM THE TABLE
KEEP UPDATING THE PROBABILITY LIKE IN THE COMPRESSION PHASE, NEXT SYMBOL “L” AFTER THE PREVIOUS SYMBOL “E” ALREADY EXISTS AND THE PROBABILITY STILL 100%
IF WE HAVE A HIT HERE BUT THE ALGORITHM DOES NOT KNOW IF IT SHOULD CHOOSE “L” OR “O”
NEXT SYMBOL “L” AFTER THE PREVIOUS SYMBOL “L” ALREADY EXISTS BUT NOW WE SAW “L” THREE TIMES AND “O” JUST ONCE, SO THE PROBABILITIES OF SYMBOLS FOLLOWING “L” ARE 66% OF ANOTHER “L” AND 33% OF AN “O”
PROB TABLE
null 100 H
H 100 E
E 100 L
L 50 L
50 O
O 100 null
null 100 H
1 == HIT
1 == HIT
1 == HIT
0 == MISS
L 66 L
33 O
1001000
H
1000101
E
1001100
L
1001100
L

Figure 17. Continuing the decompression after the learning phase: The table is still being updated with new probabilities, but now the algorithm has enough knowledge about our data to decompress a one-bit symbol with total precision. Note that to decompress the second “L,” if we had a hit before the algorithm could guess “L” or “O” because both had the same probability of 50%, from now on the algorithm will correctly predict “L” because “L” has greater probability (66% against 33%).

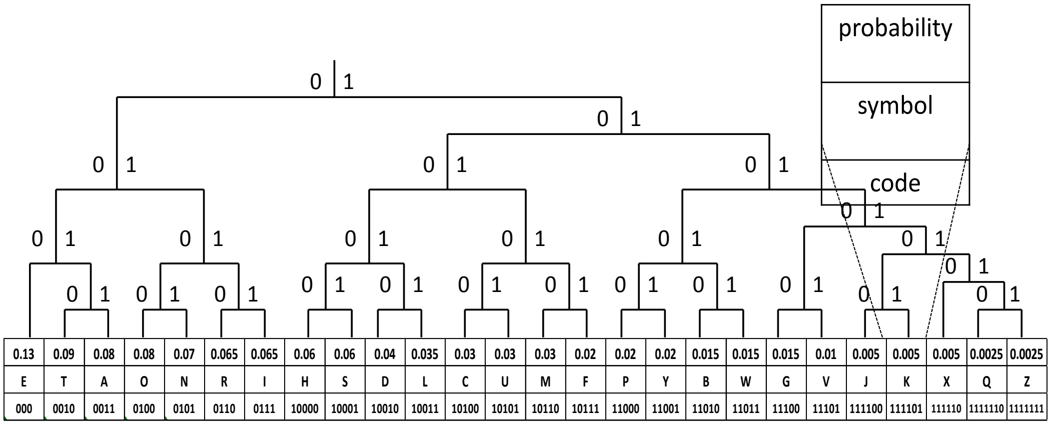
Other prediction-based methods compare the prediction with the original symbol and output the difference between them. Therefore, if the prediction is an exact match, the output is a “zero” representation; if they have a partial match, a “difference” representation is the output. It is usually smaller than the original data. David Solomon describes how this difference can compress data: “…the differences tend to be distributed according to the Laplace distribution, a well-known statistical distribution, and this fact helps in selecting the best variable-length codes for the differences” [8]. However, the table complexity increases with this method, because now we store the previous symbol, the next probable symbols, their probabilities, and a code for that symbol.

### *a. Huffman Coding*

The Huffman algorithm assigns codes to symbols and replaces the original symbol by its respective code in the compressed string. Compression is achieved because the codes are variable in length, and the shorter codes (one bit is the smallest) are assigned to the most frequently used symbols. The longer codes are assigned to the least frequently used symbols.

For frequently used symbols, the representation for each symbol, can be reduced to one bit. If we are using an eight-bit representation, this results in an 8:1 compression ratio for the most frequent symbol, which increases the ratio for the overall string.

The Huffman algorithm implements this code using a binary search tree as described in Figure 18. It presents the Huffman code implemented in a binary tree for the English alphabet using the frequency with which the letters appear in English text [9].



| | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.13 | 0.09 | 0.08 | 0.08 | 0.07 | 0.065 | 0.065 | 0.06 | 0.06 | 0.04 | 0.035 | 0.03 | 0.03 | 0.03 | 0.02 | 0.02 | 0.02 | 0.015 | 0.015 | 0.015 | 0.01 | 0.005 | 0.005 | 0.005 | 0.0025 | 0.0025 |
| E | T | A | O | N | R | I | H | S | D | L | C | U | M | F | P | Y | B | W | G | V | J | K | X | Q | Z |
| 000 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 10000 | 10001 | 10010 | 10011 | 10100 | 10101 | 10110 | 10111 | 11000 | 11001 | 11010 | 11011 | 11100 | 11101 | 111100 | 111101 | 111110 | 1111110 | 1111111 |

Figure 18. Huffman code for the 26-letter alphabet. The algorithm orders the symbols by their frequency in English text and allocates the positions in the tree with a smaller code assigned to the most frequent symbols.

The code has some restrictions: the smallest possible code has one bit, so if the alphabet has one-bit symbols (only two symbols, “0” and “1” in this case) like monochromatic images, we have to incorporate these symbols into a larger bit representation in order to represent additional characters, and then treat the combination as our alphabet [9].

#### *b. Adaptive Huffman*

Huffman coding assumes that the frequency of symbols is known by the algorithm, which is not true for some kinds of data. Some texts may not obey the normal distribution of letters, so in many implementations, the algorithm has to read data twice, slowing down the process. The first pass is used to calculate frequencies and build the tree, then a second pass is used to compress the data. This is called adaptive Huffman code. The implementation of the UNIX “compact” program [9] uses adaptive Huffman codes, for example.

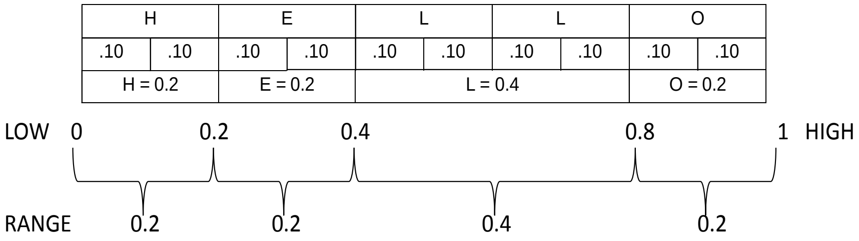
The adaptive code starts with an empty tree and fills it as data is read and compressed. Each time a new symbol is read, the algorithm computes a new frequency for that symbol and updates the tree. This method works if the decompression algorithm starts with an empty tree also, and updates the tree in the same way the compression algorithm does, following the same principles as in our example of a statistical table. We can follow the explanation in [9]: “The first symbol being input is simply written on the output stream in its uncompressed form. The symbol is then added to the tree and a code assigned to it. The next time this symbol is encountered, its current code is written on the stream and its frequency incremented by one. Since this modifies the tree, it (the tree) is examined to see whether it is still a Huffman tree (best codes). If not, it is rearranged, which results in changing the codes. (…) The decompressor mirrors the same steps. When it reads the uncompressed form of a symbol, it adds it to the tree and assigns it a code. When it reads a compressed (variable-length) code, it scans the current tree to determine what symbol the code belongs to, and it increments the symbol’s frequency and rearranges the tree in the same way as the compressor.”

Huffman methods do not have the best compression ratios because they assign codes with integer numbers to represent frequencies that are, in fact, decimal numbers, as explained in [9]: “Information theory shows that a symbol with probability 0.4 should ideally be assigned a 1.32-bit code, since −log2 0.4 ≈ 1.32. The Huffman method, however, normally assigns such a symbol a code of 1 or 2 bits,” so arithmetical methods were developed.

### *c. Arithmetic Coding*

Arithmetical codes calculate a symbol’s frequency by counting how many times it appears in the string, representing the frequency with a more complex, yet more effective, code format.

For a given string, after counting the number of occurrences of each symbol in the string, the algorithm calculates the probability by dividing its frequency by the string size (frequency / string size). The result is some number between “0” and “1” (0% and 100%). The algorithm defines three variables: “LOW”, “RANGE,” and “HIGH,” then divides the overall probability within this range [0, 1) among all symbols according to their probabilities. Figure 19 shows the base probabilities and the values of LOW, RANGE, and HIGH for each symbol.



| H | | E | | L | | L | | O | |
|---|---|---|---|---|---|---|---|---|---|
| .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 | .10 |
| H = 0.2 | | E = 0.2 | | L = 0.4 | | | | O = 0.2 | |

Figure 19. Base Table for Arithmetic coding of “HELLO.” The input string “HELLO” is read, and a probability is assigned to each symbol. Then LOW, RANGE, and HIGH values are calculated for each symbol, where RANGE = HIGH–LOW. Starting with symbol “H,” LOW=0, RANGE=0.2, and HIGH=0.2.

For symbol “E,” LOW=0.2, RANGE=0.2, and HIGH=0.4, and so on.

The code then starts as [LOW, HIGH) = [0, 1), and the following formulas are applied for each character in turn. :

- NewHigh:=OldLow+Range*HighRange(X);
- NewLow:=OldLow+Range*LowRange(X);

where Range=OldHigh−OldLow, and LowRange(X), HighRange(X) indicate the low and high limits of the range of new symbol X, respectively, from the base table.

In our exemple:

NewHigh:=OldLow+Range*HighRange(H) => NewHigh:=0+(1-0)*0.2(X) == 0.2
NewLow:=OldLow+Range*LowRange(H) => NewLow:=0+(1-0)*0(X) == 0
NewHigh, NewLow == [0, 0.2)

A good way to understand the process is to imagine that the new interval [0, 0.2) is divided among the four symbols of our alphabet using the same proportions as for the original interval [0, 1). The result is four subintervals [0, 0.04), [0.04, 0.08), [0.08, 0.16), and [0.16, 0.2). When the next symbol “E” is input, the second of those subintervals, [0.04, 0.08), is selected, as shown in Figure 20, and again divided into four subintervals [2].

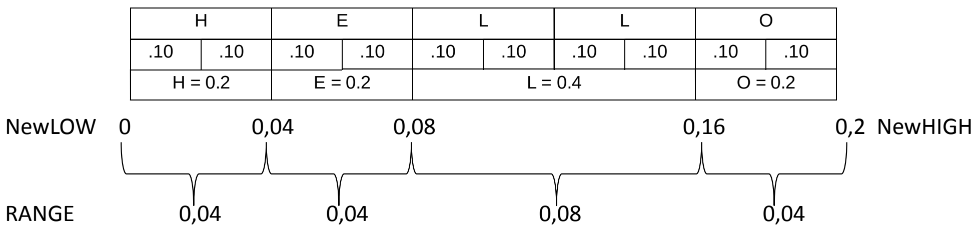


Figure 20. After the first interaction with the formulas for letter H, new probabilities are assigned to each symbol, inside the range from the previous interaction. The LOW, RANGE, and HIGH values are calculated again for each symbol. Starting

with symbol “H”, LOW=0, RANGE=0.04, and HIGH=0.04; for symbol “E”, LOW=0.04, RANGE= 0.04, and HIGH=0.08, and so on.

Using the formulas we get the same result:

NewHigh:=0+(0.2-0)*0.4(E) == 0.08

NewLow:=0+(0.2-0)*0.2(E) == 0.04

[0.04, 0.08)

This process is repeated until the last symbol is encoded:

NewHigh:=0.04+(0.08-0.04)*0.8(L) = 0.072

NewLow:=0.04+(0.08-0.04)*0.4(L) == 0.056

[0.056, 0.072)

NewHigh:=0.056+(0.072-0.056)*0.8(L) == 0.0688

NewLow:= 0.056+(0.072-0.056)*0.4(L) == 0.0624

[0.0624, 0.0688)

NewHigh:=0.0624+(0.0688-0.0624)*1(O) == 0.0688

NewLow:= 0.0624+(0.0688-0.0624)*0.8(O) == 0.06752

[0.06752, 0.0688)

Then we get the last LOW (0.06752), and remove the integer part as the final code representation of this string (06752).

To decompress we do the inverse: we get the code (06752), recognize that the original value was (0.06752), and read the code “0.06752,” which is inside the RANGE of “H” [0, 0.2). We then output “H” and apply the following formula [2] to eliminate the effect of symbol “H” from the code:

- Code:=(Code-LowRange(X))/Range

where Range is the width of the sub range of X.

Code:=(Code-LowRange(H))/Range =>Code:=( 0.06752-0.0)/0.2 == 0.3376

Then we get the code (0.3376), recognize that “0.3376” is inside the RANGE of “E” [0.2, 0.4), output “E,” and apply the formula again to eliminate the effect of symbol “E” from the code, and so on until the code achieves a value of “0,” meaning the end of the compressed string:

Code:=(Code-LowRange(E))/Range =>Code:=( 0.3376-0.2)/0.2 == 0.688
0.688 represents the range of “L” [0.4, 0.8), output “L”

Code:=(Code-LowRange(L))/Range =>Code:=( 0.688-0.4)/0.4 == 0.72
0.72 represents the range of “L” [0.4, 0.8), output “L”

Code:=(Code-LowRange(L))/Range =>Code:=( 0.72-0.4)/0.4 == 0.8
0.8 represents the range of “O” [0.8, 1), output “O”

Code:=(Code-LowRange(O))/Range =>Code:=( 0.8-0.8)/0.2 == 0
0 represents the end of the string

### *d. Adaptive Arithmetic Coding*

Like the Huffman code, the arithmetical code needs a frequency table before it starts encoding the string. Applying the same principle as adaptive Huffman, it is easy to understand the principle behind adaptive arithmetic coding.

The method starts with a 100% probability of seeing the first symbol and updates the probabilities as new symbols appear. In this algorithm, compression is made in two steps: an arithmetic-encoder step and a probability-calculation step. It reads the input stream and executes a normal arithmetic encoder. The change is that, after encoding, it updates the symbol probability table using the old counts, not the updated ones. Only after the overall process completes is the symbol count updated for the next round. The importance of updating the symbol count only after encoding and calculating the probability is that it makes it possible for the decoder to perform the inverse operation. The decoder does not know which symbol will result from the operation and cannot search any table for its probability before the overall decoding process. So it first applies the mathematical formula, compares it to the frequency range, extracts the respective symbol from it, and then, knowing what the symbol is, searches the probability table and updates the count for the next round.

## E. CRYPTOGRAPHY

### 1. Definition of Cryptography, Basic Principles and Description of General Aspects Related to Cryptography.

Throughout history, safe and secure communication has been essential. In 5 BC, the Spartans employed a cryptographic device to send and receive secret messages. This device was a cylinder called a scytale that was in the possession of both the sender and recipient of the message. Today, cryptography is needed to protect the Internet and a wide variety of network applications used in all aspects of human life. For example, the exchange of sensitive personal information such as credit-card numbers through the Internet is a common practice. Thus, protecting data and all related electronic systems is crucial, and cryptography plays a significant role.

Cryptography is from the Greek word “kriptographia” (κρυπτογραφία), literally, “secret write” or the art of writing secrets. According to [18], the definition of cryptography is: “the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques.”

A secret method of writing is called a cipher [20], in which the cleartext, or plaintext, is transformed into ciphertext. This process is called encryption, while the reverse is decryption. A key is required to encrypt or decrypt. In symmetrical cryptography, the same key is used for both. In asymmetrical cryptography, different keys are used. The encryption / decryption process is depicted in Figure 21. Another essential cryptographic primitive is the cryptographic-hash function, which transforms a variable-length input string into a fixed-length output digest. Cryptographic-hash functions must be one way, meaning that it is very hard to determine the input given the output, and collision-resistant, meaning that it is very hard to find two inputs that result in the same output.

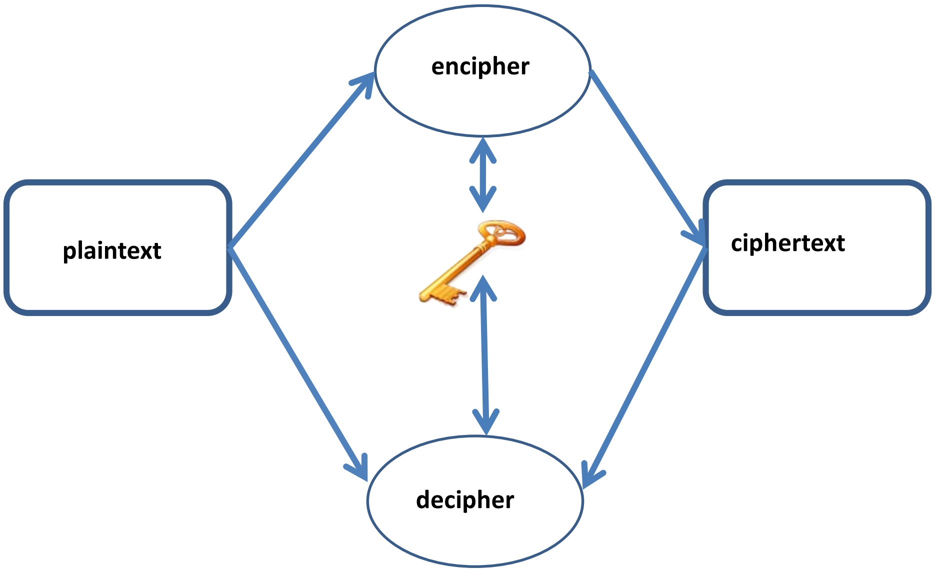


Figure 21. Secret writing, the process by which plaintext is transformed into ciphertext (After [20]). Transforming plaintext to ciphertext is called encryption, while the reverse is called decryption. A key is required to encrypt or decrypt.

## 2. Cryptographic Services

Cryptography is essential to a variety of electronic platforms such as virtual private networks, electronic commerce, wireless phones, data communications, and smart cards. A well-defined and implemented cryptosystem should provide the following services [18]:

- Confidentiality ensures the prevention of unauthorized data observation. Confidentiality can be achieved through encryption and decryption algorithms.
- Data integrity prevents the unauthorized modification of data. A useful method of enforcing data integrity is cryptographical hash functions.
- Authentication is the process that verifies the identity of a specific entity involved in a communication session. There are two kinds of authentication. Entity authentication focuses on the authentication process between the entities of a communication session; data-origin authentication is responsible for the authentication of

the origin (time, owner) of data transmitted and received during a communication session [19].

- Non-repudiation is a process that prevents an entity from denying that he sent data during a specific communication session or, in general, prevents the denial of the authenticity of data, mail, or digital signatures transmitted during a communication session.

### 3. A Basic Scenario of Cryptographic Application

In order to understand how cryptography applies to secure communication we present the following simple communication scenario [19], which is depicted in Figure 22. Assume that there are two participants, Alice and Bob, and they intend to communicate. Also, a third party, Eve, is an eavesdropper. If Alice wants to send a message to Bob, she encrypts the plaintext using a cipher that she and Bob have agreed upon. While Eve may be aware of the encryption method, she does not know the key. Kerckhoff's principle states that a cryptosystem should be secure even if the design of the cipher is public. In order for Bob to decrypt the received message, he uses the decryption key. Eve might have the following goals:

- Get and read that message.
- Retrieve the key and thus decrypt all messages encrypted with that key.
- Alter Alice’s message or replace it with another.
- Masquerade as Alice and communicate with Bob, while Bob thinks he is communicating with Alice.

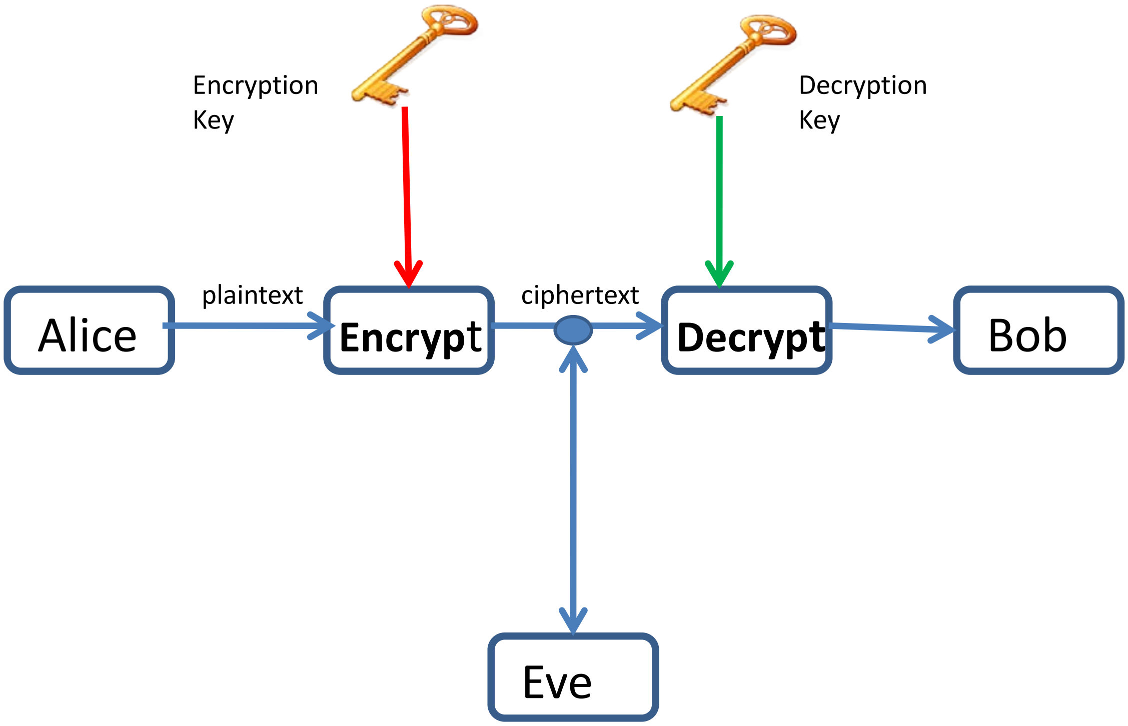


Figure 22. Fundamental Communication Scenario for Cryptography (After [19]). The two entities, Alice and Bob, want to communicate with each other. For Alice to send a message to Bob without Eve's access, she uses an encryption key to encrypt her message (plaintext). Bob receives the encrypted message (ciphertext), and decrypts it with a decryption key.

There are four types of attack [19] that Eve can implement. These attacks are based on the amount of information available to Eve for cryptanalysis.

- Ciphertext only
- Known plaintext, where she has both the ciphertext and corresponding plaintext.
- Chosen plaintext, where she has temporary access to the encryption cipher, but cannot retrieve the key. However, she can encrypt a chosen plaintext and try to determine the key.
- Chosen ciphertext, where she has temporary access to the decryption mechanism and tries to determine the key.

## 4. General Description of Cryptographic Algorithms (Symmetric, Asymmetric, Hash Functions)

As stated in [20], a cryptographic algorithm, or cipher, is a mathematical function used for encryption and decryption. According to [22], cryptographic algorithms are divided into three main categories: private (symmetric) key encryption, public (asymmetric) key encryption, and hash functions.

### *a. Symmetric, Private-Key Encryption*

Symmetric-key cryptography is a class of algorithms that allows parties to communicate securely only when they share some prior secret, such as the secret key. Each user must trust the other not to reveal the key to a third party. The sender and the recipient can encrypt and decrypt a specific message using the same secret key. Figure 23 depicts symmetric-key encryption.

There are many symmetric-key algorithms, including DES (Data Encryption Standard), Triple DES, and AES (Advanced Encryption Standard). In this thesis, the 128-bit AES algorithm, utilized by traditional 2D cryptographic coprocessors [HSSEC], is incorporated into our proposed 3DIC compression-crypto (data transformation) coprocessor.

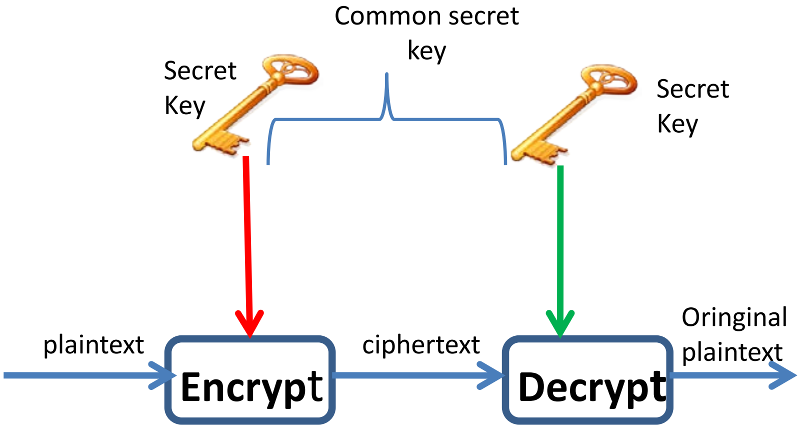


Figure 23. Symmetric key encryption (After [19]). The two entities that want to establish a communication session have already exchange shared a common secret key. The sender encrypts the plaintext with the common secret key, and the receiver decrypts it with the common secret key that he has on his possession in order to access the received message (ciphertext).

### *b. Stream and Block Ciphers*

Symmetric-key encryption is further divided into two categories [20]. The first category is the stream cipher or stream algorithm, where the encryption operates on plaintext bit by bit or sometimes byte by byte. The second category is the block cipher or block algorithm, where each block is a specific group of bits of plaintext data (64 or 128 bits), which enciphers to a ciphertext of the same length. Block ciphers have five possible modes of operation: electronic codebook (ECB), cipher-block chaining (CBC), cipher feedback (CFB), output feedback (OFB), or counter (CTR). Table 1 summarizes the modes of operation of block ciphers. A detailed discussion of the electronic codebook

(ECB) and cipher-block chaining (CBC) modes, which are going to be used for the AES-128 encryption component of the proposed cryptographic coprocessor, will follow in Chapter IV.

| Mode | Description | Typical Application |
|---|---|---|
| Electronic Codebook (ECB) | Each block of 64 plaintext bits is encoded independently using the same key. | • Secure transmission of single values (e.g., an encryption key) |
| Cipher Block Chaining (CBC) | The input to the encryption algorithm is the XOR of the next 64 bits of plaintext and the preceding 64 bits of ciphertext. | •General-purpose block oriented transmission<br>• Authentication |
| Cipher Feedback (CFB) | Input is processed j bits at a time. The preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce the next unit of ciphertext. | • General-purpose stream-oriented transmission<br>• Authentication |
| Output Feedback (OFB) | Similar to CFB, except that the input to the encryption algorithms is the preceding DES output. | • Stream-oriented transmission over noisy channel (e.g., satellite communication) |
| Counter (CTR). | Each block of plaintext is XORed with an encryption counter. The counter is incremented for each subsequent block. | •General-purpose block oriented transmission<br>• Useful for high-speed requirements |

Table 1. Block Ciphers mode operation summary (From [23]).

### *c. Public (Asymmetric) Key Encryption*

Unlike symmetric cryptography, asymmetric cryptography uses a different key for encryption and decryption. Each user must have two keys, one private and one public. The keys are mathematically related and created with a key-generation algorithm. Both the sender and receiver will keep their secret key confidential and allow their public keys to be distributed. In order for the sender and receiver to communicate, the sender must encrypt the message with the receiver’s public key. The receiver decrypts the received message with his private key. The public-key encryption process is depicted in Figure 24. According to [22], asymmetrical algorithms are poorly suited for encrypting large messages because they are relatively slow. However, these algorithms support authentication, integrity, and non-repudiation, and they allow parties who have never met to securely communicate. There are several asymmetric algorithms such as RSA, DH, Pohlig-Hellman, El Gamal, and ECC. Asymmetrical algorithms also support digital signatures, key transport, and key agreement.

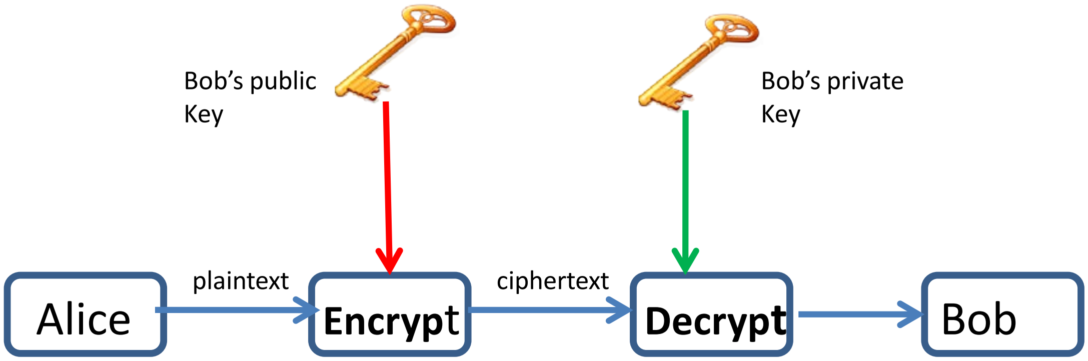


Figure 24. Asymmetric-key encryption (After [19]). Asymmetric cryptography uses a different key for encryption and decryption. Each entity (Alice and Bob) must have two keys, one private and one public. Both sender and receiver keep their secret key secret and allow their public keys to be distributed. In order for the sender and receiver to communicate, the sender (Alice) must encrypt the message with the receiver (Bob's) public key. To decrypt the received message, the receiver must use his private key. The public and private keys of each entity are related but unequal.

### *d. Hashing Functions*

A cryptographic-hash function takes a variable-length input and produces a fixed-length output using a one-way mathematical function. More precisely, as described in [18], "a hash function h maps bit-strings of arbitrary finite length to strings of fixed length, e.g., *n* bits. For a domain D and range R with h: D→R and |D| > |R|, the function is many to one, which implies that the existence of collisions (pairs of inputs with identical output) is unavoidable." The result is called a message digest, or hash code, hash-result, hash-value, or simply, hash. The output can be considered a fingerprint of the data. Using this primitive, the integrity of data can be enforced. It is feasible for any entity to reproduce the message digest from the same stream of data, but it is not feasible to create a different stream of data that produces the same message digest [22], which would result in a hash collision.

Hash functions enhance data integrity by supporting digital signature schemes, where a message is typically hashed first, then the hash-value, as a representative of the message, is encrypted [22]. Hash functions do not provide confidentiality and non-repudiation [22]. There are many cryptographical hash algorithms, including SHA-1, SHA-2, SHA-128, SHA-256, SHA-512, MD-2, MD-4, and MD-5. SHA-1 is the most widely used of the existing SHA hash functions, and it is employed in several widely used security applications and protocols. A detailed discussion of SHA-1 and SHA-512 follows in Chapter IV.

# III. COMPRESSION

## A. INTRODUCTION

Processor designers must balance several competing constraints. Compression is a useful technique for reducing power consumption and execution time, e.g., by ensuring that the most frequently fetched instructions have the shortest length in bits. In this chapter we present different algorithms and hardware architectures for compression.

## B. COMPRESSION

The main taxonomy for compression divides algorithms into lossy or lossless. Lossy compression works by throwing away information that does not substantially affect the message's ability to be understood; but this method is unable to reconstruct the original data exactly. Lossless compression, on the other hand, does not discard any information; it is merely represented by fewer bits. Decompression restores the original data exactly. Both forms of compression have a wide range of applications such as communication and computing.

Another type is "cascaded compression," where the data passes through different compression algorithms in series. This technique works well with lossless algorithms, but, with lossy algorithms, can magnify errors during decompression.

Video or audio compression can be performed using lossy compression algorithms since a small or even imperceptible loss fidelity can often be sacrificed for a substantial size reduction.

Compression performance is measured in terms of compression ratio, which is defined as the size of the output data divided by the size of the input data. A value smaller than one means that the compression yielded a size reduction. The inverse of the compression ratio is the compression factor.

In our system, we are proposing to send compressed execution traces from an untrusted computer to a trusted one, in order to reconstruct the machine's state exactly for analysis. Therefore, we require a lossless compression scheme.

Before describing algorithms suitable for our proposed system, we provide a brief survey of compression methods.

## C. COMPRESSION ALGORITHMS

Of the many compression algorithms, we focus on the methods preferred by industry (e.g., AHA and IBM) and research groups (e.g., dictionary and statistical).

## D. STRING COMPRESSION

Data can be compressed either symbol by symbol or one string at a time. Since symbols have different probabilities of being used, Huffman coding assigns a code to each symbol and compresses data symbol by symbol. Compressing strings of symbols achieves better compression ratios because a group of *n* individual symbols with different probabilities requires more bits per symbol to represent than assigning Huffman codes to the $2^n$ strings (all possible permutations) formed by the individual symbols. Dictionary methods compress strings, which one reason they are used more often than Huffman is coding and its variants [9]

## E. DICTIONARY METHODS

Dictionary compression accepts data as input and stores it in a special structure called a dictionary, outputting a pointer to its location (i.e., a token). The pointer is smaller than the data stored to that location; therefore, substituting the original data with its pointer achieves compression. Chapter II described dictionary compression, and we now present a variety of algorithms that perform dictionary compression.

### 1. Lempel Ziv

“Having one’s name attached to a scientific discovery, technique, or phenomenon is considered a special honor in science. Having one’s name associated with an entire field of science is even more so. This is what happened to Jacob Ziv and Abraham Lempel. In the late 1970s these two researchers developed the first methods, LZ77 and LZ78, for dictionary-based compression. Their ideas have been a source of inspiration to many researchers who generalized, improved, and combined them with RLE and statistical methods to form many popular lossless compression methods for text, images, and audio” [9].

### 2. LZ77

Lempel and Ziv 77 [LZ77 also known as LZ1] is a dictionary method that uses a sliding window for search, and a look-ahead buffer, as presented in Chapter II. The look-ahead buffer is divided into three parts: the offset (distance between the symbol being encoded and the same symbol previously seeing), the length of the matching string, and the next symbol to be read. The offset size is the $\log_2$ of the length of the search buffer, which may be a few thousand bytes long. Therefore, the offset size is about ten to twelve bits. The length part is the $\log_2$ of the length of the look-ahead buffer (L, which is on the order of tens of bytes) minus one (for the next symbol field): $\log_2(L - 1)$, which results in a field a few bits long. The next symbol field is about 8 bits and depends on the size of the alphabet used. Therefore, the total size of the token is about 11 + 5 + 8 = 24 bits, and the encoder needs to encode a string with a size of at least 3 bytes (24 bits) at a time, in order to not produce a larger file than the original.

The most difficult part of the algorithm is the search. Every search implementation must balance between speed and memory size. “A binary search tree gives good performance for only modest memory requirements [24],” but “a hash table appears to be a contender [24],” because many hash tables are available for improving performance.

### 3. LZR

LZR derives from the LZ77 method, but the lengths of buffers are unbounded [9]. It will search the entire space for the best match, and it incurs a memory overhead to store data, as well as a time overhead for searching this data structure. It manages the memory space by increasing the size of the buffers until no more space is available and maintaining them at that size, or by deleting the buffers and starting to fill the memory again.

To reduce the search time, LZR proposes the use of a suffix tree: instead of deleting nodes and recomposing the tree’s structure, just mark the nodes as deleted, and delete the tree if all nodes in a tree are marked as deleted. Another drawback of this method is the size of the output, because of the unbounded buffers, which are reduced by encoding the output using a variable-length code that allows reduction of the output from order $n$ to order $2\log_2 n$.

## 4. LZSS

Storer and Szymanski developed LZSS in 1982 [26]. LZSS is also a variant of LZ77 but with several improvements: the implementation of the look-ahead buffer in a circular queue; the implementation of the search buffer in a binary search tree; and the output token's reduction to two fields instead of three (only the offset and the length are present). The LZ77 representation for "no match" is a token with the three fields (0, 0, next symbol); in LZSS a one-bit flag is set for every output. If no match is found, LZSS outputs the one-bit "miss" flag and the original uncompressed data. If a match is found, LZSS outputs a one-bit "hit" flag followed by the two-field token. "LZSS decoding is very fast and comparatively little memory is required for encoding and decoding" [26].

LZSS also changed the size of the buffers to output a one-byte uncompressed ASCII, or a two-byte compressed token; the search buffer has two kilobytes (= $2^{11}$), and the look-ahead buffer has 32 bytes (= $2^5$). This results in an offset field of eleven bits and a length field of five bits, for a total of two bytes (remember that token fields are a representation of the actual data, so we need $n$ bits to represent $2^n$ bits of data). For the extra one-bit flag, they collect and output those flag bits eight at a time for the next eight items (uncompressed ASCII or tokens), for performance reasons.

A significant change that improves speed is the implementation of a binary search tree for the search buffer. This buffer has a fixed size: the tree grows until the maximum size is achieved and keeps the same number of nodes, changing only its shape, according to the deletion and insertion of nodes in different places, until the end of the compression, when the tree decreases in size until no more symbols are stored and the tree becomes empty.

The algorithm for LZSS is the following [26]:

```
while lookahead buffer not empty do
        get a pointer (offset,length) to the longest match in the window for the
        lookahead buffer
if length > p then
        output the pointer (offset, length)
        shift window length characters
else
        output first character in lookahead buffer
        shift window one character
```

## 5. LZB

LZB, proposed in 1987, is an improvement over LZSS and changes the token's representation. It adapts the size of the first field of the token (the offset) according to the actual size of the search buffer being represented. At the beginning of compression, if just two symbols are stored in the buffer, only one bit is needed to represent them with the token. The number of bits increases as the buffer is filled, and after 50% of the buffer is filled, the maximum number of bits is used. For the second field of the token (the length), the algorithm uses the Elias gamma code, a variable-length code, to represent small numbers with fewer bits. In this code, the length of the integer $n$ is $1 + 2\ |\log_2 n|$, and it is ideal for cases where $n$ appears in the input with probability $1/\ (2n^2)$. The Elias gamma code can represent small numbers with fewer bits than a fixed-size code. The example below, adapted from [9], summarizes this code. Figure 25 shows a graph of its performance.

For the number $n$=20, the encoding occurs as follows:

1. Find the largest integer $N$ such that $2^N \leq n < 2^{N+1}$ and write $n = 2^N + L$.

   *$2^N \leq n < 2^{N+1} \Rightarrow 16 < 20 < 32 \Rightarrow 2^4 < 20 < 2^5 \Rightarrow N=4$*

   *$n = 2^N + L \Rightarrow 20 = 2^4 + L \Rightarrow L=4$* Notice that $L$ is at most an $N$-bit integer.
2. Encode $N$ in unary either as $N$ zeros followed by a 1 or $N$ ones followed by a zero.

   *N=4 = 00001 in unary (four 0s followed by 1)*
3. Append $L$ as an $N$-bit number to this representation of $N$.

   *L=4 = 0100 in N-bit binary, so 00001 append 0100 = 000010100*
4. $n$ = 000010100

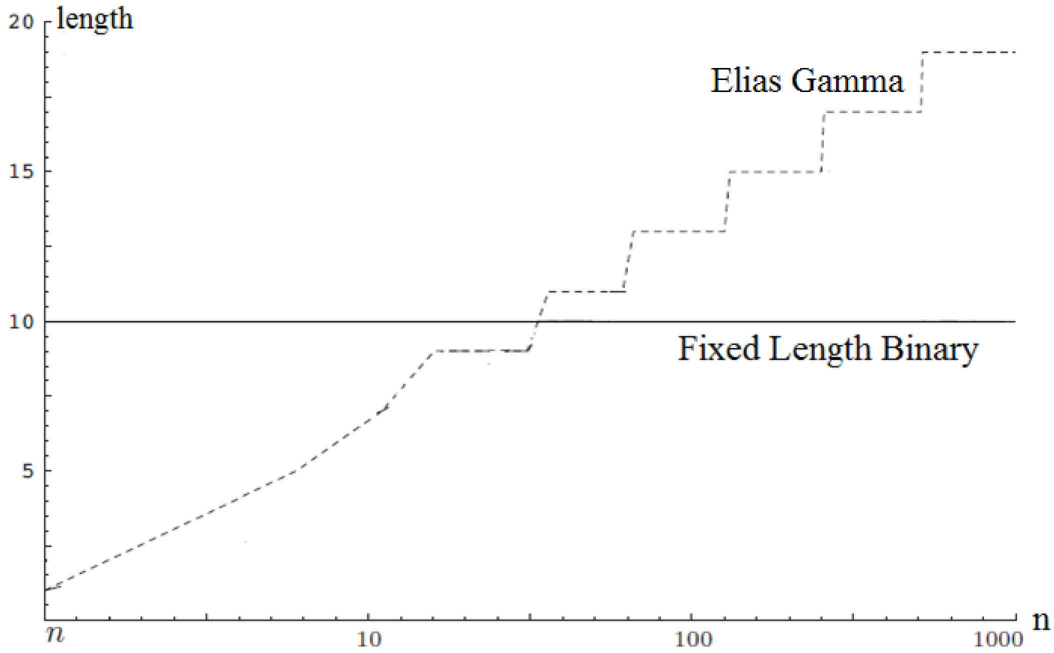


Figure 25. This graph shows the lengths of the Elias gamma code and the standard variable length binary (beta) code, comparing them and showing the advantage of Elias gamma codes for small numbers *n* (From [9]).

### 6. GZIP

GZIP is one implementation of Deflate (a "public domain compression method based on a variation of LZ77 combined with Huffman codes" [9]), which was developed by Philip Katz and implemented in PKZIP, supporting the zip-file format and other variants. In GZIP, the token has two fields, as with the previous method: the offset, limited to 32K bytes, and the length, limited to 258 bytes. As before, if no match occurs, the uncompressed string is written to the output.

In contrast, Huffman codes are written for the token fields, using two tables because of the different sizes of the output fields: one for lengths (limited to 258 bytes) and uncompressed strings (bytes normally in the interval 0-255), and another for offsets (up to 32 Kbytes). When an offset/length is found, the algorithm searches the tables for its Huffman codes.

The algorithm also can perform two searches and compare them in order to find the longest match in three different modes:

1. The default mode (the longest-length match): the first match is compared with a predefined value. If the length is greater than this value, a second search is not done, and if it is smaller, a second search is done.
2. If the user desires speed: a second match is not done, which comes at the expense of a poor compression ratio.
3. If the user desires the best compression ratio: a second search is always done over the entire search buffer, at the expense of more time.

Another improvement is the practice of generating two different kinds of tables: a fixed table built into the encoder and decoder, used to speed up the process, which may not be optimal for compressing certain types of data; and a flexible table constructed from statistical data collected at runtime from the data being compressed. The disadvantage of this approach is that those tables have to be compressed together with the data in order to allow the algorithm to perform the decompression. These runtime-generated tables are Huffman encoded; therefore, along with the data, the output file has a Huffman table for decoding the two encoded tables.

Due to the importance of GZIP, other deflate variants, and this new table structure, we investigate more deeply how the fixed, built-in tables represent the data.

Length/uncompressed tables have pre-codes from 0 to 285, which may be followed by extra bits. From 0 to 255, the pre-codes are used to represent uncompressed literals; pre-code 256 is used to represent end of block; and pre-codes from 257 to 285 represent lengths (figure 26a). Rather than forming the output string, they are used as references to the length-code table (figure 26b), which will convert them in the actual output-code bits. For the offset, a table with five-bit, fixed-length codes and extra bits are used to represent all 32,768 possible offsets (represented in decimal instead of binary in Figure 26c.

a)

| Length table | | |
|---|---|---|
| Pre-Code | Extra bits | Lengths |
| 257 | 0 | 3 |
| 258 | 0 | 4 |
| 259 | 0 | 5 |
| 260 | 0 | 6 |
| 261 | 0 | 7 |
| 262 | 0 | 8 |
| 263 | 0 | 9 |
| 264 | 0 | 10 |
| 265 | 1 | 11,12 |
| 266 | 1 | 13,14 |
| 267 | 1 | 15,16 |
| 268 | 1 | 17,18 |
| 269 | 2 | 19–22 |
| 270 | 2 | 23–26 |
| 271 | 2 | 27–30 |
| 272 | 2 | 31–34 |
| 273 | 3 | 35–42 |
| 274 | 3 | 43–50 |
| 275 | 3 | 51–58 |
| 276 | 3 | 59–66 |
| 277 | 4 | 67–82 |
| 278 | 4 | 83–98 |
| 279 | 4 | 99–114 |
| 280 | 4 | 115–130 |
| 281 | 5 | 131–162 |
| 282 | 5 | 163–194 |
| 283 | 5 | 195–226 |
| 284 | 5 | 227–257 |
| 285 | 0 | 258 |

c)

| Offset table | | |
|---|---|---|
| Codes (in decimal) | Extra bits | Offset |
| 0 | 0 | 1 |
| 1 | 0 | 2 |
| 2 | 0 | 3 |
| 3 | 0 | 4 |
| 4 | 1 | 5,6 |
| 5 | 1 | 7,8 |
| 6 | 2 | 9–12 |
| 7 | 2 | 13–16 |
| 8 | 3 | 17–24 |
| 9 | 3 | 25–32 |
| 10 | 4 | 33–48 |
| 11 | 4 | 49–64 |
| 12 | 5 | 65–96 |
| 13 | 5 | 97–128 |
| 14 | 6 | 129–192 |
| 15 | 6 | 193–256 |
| 16 | 7 | 257–384 |
| 17 | 7 | 385–512 |
| 18 | 8 | 513–768 |
| 19 | 8 | 769–1024 |
| 20 | 9 | 1025–1536 |
| 21 | 9 | 1537–2048 |
| 22 | 10 | 2049–3072 |
| 23 | 10 | 3073–4096 |
| 24 | 11 | 4097–6144 |
| 25 | 11 | 6145–8192 |
| 26 | 12 | 8193–12288 |
| 27 | 12 | 12289–16384 |
| 28 | 13 | 16385–24576 |
| 29 | 13 | 24577–32768 |

b)

| Length code table | | |
|---|---|---|
| Pre-Code | bits | Codes (in binary) |
| 0–143 | 8 | 00110000–10111111 |
| 144–255 | 9 | 110010000–111111111 |
| 256–279 | 7 | 0000000–0010111 |
| 280–287 | 8 | 11000000–11000111 |

Figure 26. In the length pre-code table (a), codes from 257 to 285 are used to represent lengths; these pre-codes are used as references to the length-code table (b), which will convert them to the actual output-code bits. Table (c) shows the five-bit, fixed-length codes and extra bits used to represent all 32,768 possible offsets (After [9]).

The deflate search uses large buffers; therefore, instead of moving data, the algorithm moves a pointer to indicate where the search buffer ends and the look-ahead buffers start. The strings are hashed and stored in a hash table; the encoder then hashes the next string and compares the hashes, searching for matches.

Deflate and its variants such as GZIP are very important due to their performance, speed, and availability of free implementations: “deflate normally produces compression factors of 2.5 to 3 on text, slightly less for executable files, and somewhat more for images. Most important, even in the worst case, deflate expands the data by only 5 bytes per 32 Kb block. Also, free implementations to avoid patents are available” [9].

### 7. LZ78

LZ78 does not use any buffer or sliding window; instead, it uses all assigned memory to store previously seen strings in a dictionary structure. It starts with an empty dictionary and reads the first symbol, a one-symbol string. As LZ78 executes, the current symbol is read and the dictionary is searched for a match. If a match is found, LZ78 reads the next symbol, concatenates it with the previous, and tries to find a two-symbol match in the dictionary. This process continues until a new concatenated symbol causes a miss during the search. The algorithm then outputs a pointer to the longest match (the first field in the token), outputs the symbol that caused the miss (the second field in the token), and stores the newly formed string in the dictionary. “LZ78 and arithmetic codes outperform LZ77 in the compression achieved, and the encoding speed” [26].

Dictionary sizes can vary: a long dictionary can store more strings, but the pointers are bigger and the search process longer. Dictionaries are implemented as a tree (one that is not a binary tree); each new symbol added to a string is appended to the tree as a child of the last string’s value, as shown in Figure 27.

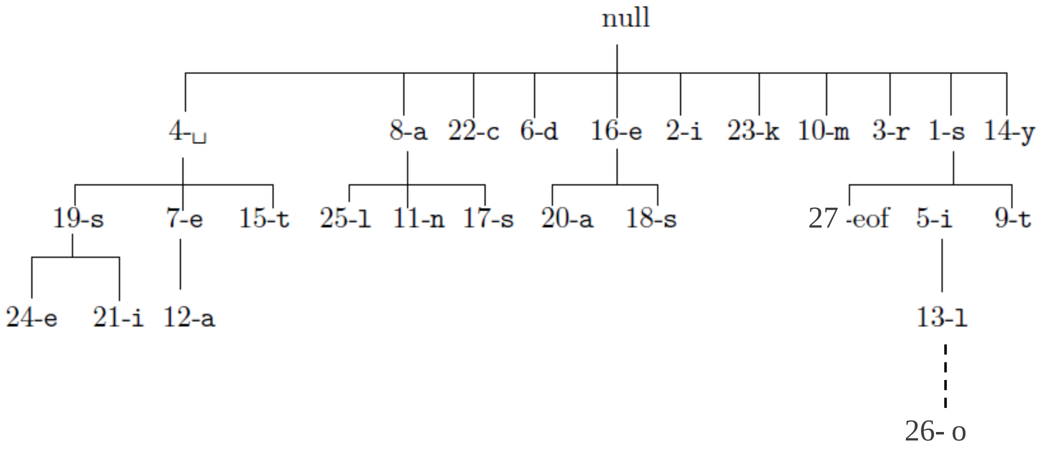


Figure 27. Implementation of the LZ78 tree (From [9]). Each new symbol is appended to the tree as a child of the string to which it belongs. For example, in the new string *silo,* the algorithm matches *sil*, outputs *(1, o)* (a pointer to the beginning of the string, new value), and *o* is appended as a child of 13-l.

## 8. LZW

LZW, published in 1984, is a variant of LZ78, in which the second field of the token is omitted. It outputs just the pointer to a dictionary location. For a normal 8-bit alphabet, the dictionary is preloaded with all 256 one-symbol strings; therefore, a match is guaranteed for the first input symbol, eliminating the need for the first (0, symbol) “miss” token. The next symbols to be appended to these one-symbol strings are inserted at positions above 257. The dictionary is implemented as a tree similar to the LZ78 algorithm, but instead of appending the new symbols as children of the strings, they are concatenated with the previous symbol or string. This new string is written to a location determined by its hash value along with its parent’s address. The values stored at this location are the new portion of the string and the address of its parent.

The decoder works in the opposite manner; it takes the same dictionary of one-symbol strings, plus the output tokens, and reads the content of each token’s location. At the beginning of the decompression phase, the first tokens correspond to one-symbol strings. After input, the first token the decoder reads the next symbol and concatenates it with those one-symbol strings, resulting in a two-symbol string. The decoder stores this string at a location pointed to by the hash value of this new two-symbol string and writes the actual one-symbol string’s location as its parent. Figure 28 shows the encoding process and Figure 29 shows decompression. The problem, not shown in the figures, is that the decoder decodes the string in reverse order, using a stack to store the string and pop the values for the output at the end of the process in the right order. LZW is slower because just one character is appended at a time.

| symbol | new string | preloaded dictionary | | | | | | | | | blank dictionary | | | | | | **token** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | … | 101 | … | 104 | … | 108 | … | 111 | 257 | 258 | 259 | 260 | 261 | 262 | |
| h | he | null | … | e | … | h | … | l | … | o | 104,e | | | | | | **104** |
| e | el | null | … | e | … | h | … | l | … | o | 104,e | 101.l | | | | | **101** |
| l | ll | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | | | | **108** |
| l | lo | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | 108,o | | | **108** |
| o | oh | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | 108,o | 111,h | | **111** |
| h | he* | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | 108,o | 111,h | | **257** |
| e | hel | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | 108,o | 111,h | 257,l | |
| l | ll* | null | … | e | … | h | … | l | … | o | 104,e | 101.l | 108,l | 108,o | 111,h | 257,l | **259** |
| l | l | null | … | e | … | h | … | l | … | o | | | | | | | |

Figure 28. LZW Encoder. The string *hellohell* is read: the first symbol *h* is preloaded into the dictionary, resulting in a match. The next symbol is *e*, and the encoder tries to find a match for the concatenation *he* (the longest match) but fails; therefore, it outputs the address for *h*. Since the concatenated string *he* is a new string, the encoder also stores this string at a new address (257) provided by a hash function; the values stored are its parent’s location (*h*=104) and the new value added (*e* in this case). The algorithm always tries to find the longest match; therefore, when the algorithm inputs the second *h*, it also inputs the second *e* as before and tries to find a match for *he*. Now that *he* is stored, the algorithm also tries the three-symbol string *hel* but fails to find a match. Therefore, after failing to match order three, the algorithm returns to order two and outputs this location (257).

| token | preloaded dictionary | | | | | | | | | **symbol** | new | blank dictionary | | | | | | **symbol** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | … | 101 | … | 104 | … | 108 | … | 111 | | | 257 | 258 | 259 | 260 | 261 | 262 | |
| 104 | null | … | e | … | h | … | l | … | o | **h** | | | | | | | | |
| 101 | null | … | e | … | h | … | l | … | o | **e** | he | 104,e | | | | | | |
| 108 | null | … | e | … | h | … | l | … | o | **l** | el | 104,e | 101.l | | | | | |
| 108 | null | … | e | … | h | … | l | … | o | **l** | ll | 104,e | 101.l | 108,l | | | | |
| 111 | null | … | e | … | h | … | l | … | o | **o** | lo | 104,e | 101.l | 108,l | 108,o | | | |
| 257 | null | … | e | … | h | … | l | … | o | | | 104,e | 101.l | 108,l | 108,o | 111,h | | **he** |
| 259 | null | … | e | … | h | … | l | … | o | | hel | 104,e | 101.l | 108,l | 108,o | 111,h | 257,l | **ll** |

Figure 29. LZW Decoder. The first token (104) is read, and the output of this address (h) is the first symbol. The process continues with the next token (101), which results in a new symbol (e), but now it also stores the concatenation (he) of the previous two symbols in the dictionary (257). Like the encoder, when the decoder reads the token 257, the dictionary already contains its value: (104, e) -> (h, e) -> he. The output is the concatenation of the two-symbol columns.

### 9. LZC (UNIX Compress)

LZC is a variant of LZW, but with a different dictionary implementation. It starts with a size of 512 entries, with the first 256 already filled as with LZW, and 9-bit pointers to those 512 locations. When the dictionary is filled up, it doubles in size, using 10 bits to represent the 1024 locations and so on, until the limit is reached. The limit can be set by the user, up to 16-bit pointers, the default being the 16-bit pointer. When the limit is reached, the dictionary becomes static, and the algorithm starts monitoring the compression ratio. If the ratio drops below the threshold value, it erases the dictionary and starts the filling process again. The decoder will detect the special symbol for this erase process, and will also erase the dictionary and start filling it again.

### 10. LZT

LZT is an improvement over LZC; the difference is the way it handles a full dictionary. The LZT algorithm stores a dictionary and a linked list of keys sorted by the number of times they are used. When the dictionary becomes full the algorithm deletes the least-recently-used (LRU) key and its reference in the dictionary.

This represents an improvement over LZC because when LCZ deletes the entire dictionary, its performance immediately drops, and in LZT, with just part of the dictionary deleted, the performance becomes more consistent. The downside is that LZT is slower because of this new structure of linked lists of keys.

### 11. LZMW

LZMW, published in 1985, is a variant of the LZW algorithm, with a difference in the dictionary implementation. The LZMW algorithm, like LZT, deletes the least-recently-used (LRU) dictionary entries, and it also must keep track of the dictionary used in an additional structure. The major difference of this method is that, unlike LZW, the dictionary can grow by more than one symbol at a time. Instead of having one symbol added to the dictionary and a second one pointing to it, both can be stored in the same dictionary location, meaning the algorithm adapts faster to the input data. However, this also introduces some problems: during the dictionary search, suppose that the dictionary already has strings *aaaa* and *aaaaaaaa*. During the search, the algorithm must go until the eighth symbol of the string *aaaaaaab* to realize that it is necessary to choose the shorter phrase *aaaa* [9].

### 12. LZFG

LZFG is a mix between LZ77 and LZ78; it outputs a mix of literals and tokens. If the output is not compressed, it outputs the “code for literal,” followed by the size of this literal. If the output is found, it outputs the “code for token,” the offset from the previously seen string to the first symbol of the compressed string, and the number of symbols to input. Figure 30 shows an example.

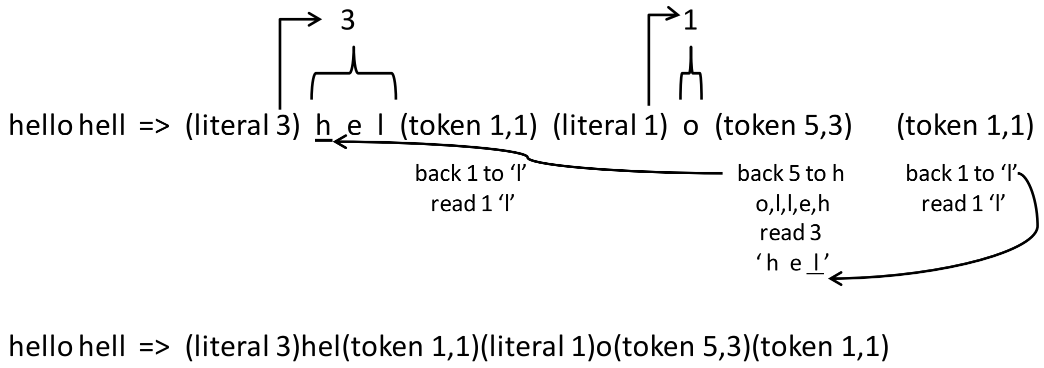

3
1
hello hell => (literal 3) h e l (token 1,1) (literal 1) o (token 5,3) (token 1,1)
back 1 to 'l'
read 1 'l'
back 5 to h
o,l,l,e,h
read 3
' h e l'
back 1 to 'l'
read 1 'l'
hello hell => (literal 3)hel(token 1,1)(literal 1)o(token 5,3)(token 1,1)

Figure 30. The compression of string *hellohell* is shown. For the first three symbols *h, e,* and *l,* there is no match, so they are output as literals. For the fourth symbol, *l,* there is a match returning one position and reading one symbol, so it is output as a (token, 1, 1). When the next tokens count back for a match, all previous symbols are included in the count, even the symbols represented by previous tokens.

## 13. ALDC

ALDC is an LZSS-based method patented by IBM and used in commercial IBM and AHA compressor coprocessors. The dictionary is filled during compression in the same way as is done during decompression using the input data. Therefore, like the other methods, it is not necessary to include the dictionary in the compressed data, but it has to be reset before each new data file is input.

Data is stored sequentially to the dictionary from location zero to the maximum available size, which can range from 512 bytes to 1024 and 2048 bytes. The increase in compression ratio is only 3% for each size upgrade; therefore, 512 bytes has the best cost/benefit configuration. The algorithm processrs data one byte at a time, and when the dictionary is full, the oldest data is replaced. The output for a hit is a one-bit flag to identify the hit, and a two-field token, the byte count (size), and the history location (address). If the string is not already stored in the dictionary, the output is a miss bit plus the literal (original data). ALDC uses a non-adaptive coding scheme to arrange the output. The data address is encoded using fixed-size binary, and the length is a quasi-logarithm code from two to twelve bits, for a total of 286 values, with the last sixteen values used for control and the last value defined as the end of compressed stream.

The 270 previous codes are for output lengths and can range from two to 271 bytes (the one-byte string is not considered since there is no compression).

### 14. Dictionary Summary

The difference between these algorithms is the way they handle the dictionary write and search process; when a match is found, a token with its position in the dictionary is output. If the token has fewer bits than the data, then compression is achieved.

## F. STATISTICAL METHODS

Statistical compression inputs data and outputs a variable-length code with short codes assigned to most used symbols, based on a statistical table. The code is smaller than the data it represents; therefore, compression is achieved by substituting the original data with its code. Statistical compression was introduced in Chapter II, and we now present some statistical compression algorithms.

### 1. Prediction

Prediction is a method in which the algorithm tries to predict the next symbol based on the context of data. Prediction-based compression algorithms can be divided as follows.

1. Statistical model: the algorithm counts the times a symbol appears in the input data and assigns probabilities based on this count. For example, if symbol *s* was seen three times in a ten-symbol data stream, its probability is 3/10.
2. Context-based statistical model: the algorithm does not count the times a symbol appears, but rather how many times it appears after a string (context) was seen.
3. Static-context-based statistical model: the string (context) that precedes the actual symbol is fixed. The algorithm has preloaded bigrams (or trigrams) of the alphabet used, but everything else stays the same.
4. Adaptive, context-based, statistical model: the context string is variable in length. An order *N* algorithm starts storing and comparing strings with length *N* and decrements the length by one symbol if no match is found, until the algorithm

reduces to order 0, meaning that just the actual symbol, without any preceding context string, is seen. If this symbol was not seen before, the algorithm assigns a probability to this symbol and stores it.

Some prediction methods use statistical data to guess the next input and compare the guess with the actual symbol. If they match, a hit symbol is output; otherwise, a miss symbol plus the original data is output, as explained in Chapter II. Other methods like PPM (explained below) use this statistical data with an arithmetic encoder.

### 2. PPM

Prediction with partial-string matching (PPM) of order *N* is an algorithm that uses a set of *N*+1 Markov predictors [3]. PPM computes the probability of a symbol and sends this symbol to an adaptive arithmetic encoder to be encoded with the following probability: the length of the encoded data in bits = $-\log_2$ (Probability). If symbol *S* was seen previously with probability ½, its encoded length is: $-\log_2 (1/2) = 1$ bit.

The algorithm reads a symbol *S* and searches the order *N* context (the length *N* string seen before *S*). If the order *N* string was seen before, followed by the symbol *S*, a probability *P* of this occurrence exists. S is sent to the encoder to be encoded with probability P, and the probability is then updated to compute this new occurrence. Otherwise, if no string that precedes *S* was found, the algorithm reduces to order *N*-1, and a new search is performed. The process continues until a string matches or it reduces to order 0, when a search is performed for *S* itself, without any context. If no match is found, it switches to an order known as order -1: *S* is stored to the probability table and encoded with probability 1. [should there be a period instead of colon after -1?]

To keep the decoder consistent with the encoder, the algorithm needs to add a flag each time the order of the context is reduced. This is accomplished using an “escape” symbol, which is written to the output each time the order is changed. If the decoder recognizes the escape symbol, it switches to the same order; the arithmetic encoder also encodes the escape symbol with its probability. The worst case occurs when no match is found: the encoder reduces to order -1, outputting *N*+1 escape code along with the original data (arithmetically encoded with probability 1, the size of the alphabet).

### 3. PPMA

This method differs from PPM in the way it encodes the escape symbol. Instead of just sending it to the arithmetic encoder, it fixes the escape symbol's probability at 1/ (*N*+1), which is equivalent to assigning it the same count of 1 every time. Also, the probability of the escape symbol is not computed together with the data probability.

### 4. PPMB

The main idea of PPMB is to store only contexts that appear more than one time; therefore, PPMB only updates the symbol probabilities after encountering the same symbol twice. PPMB accomplishes this by subtracting 1 from the symbol count.

### 5. PPMC

PPMC keeps track of all comparisons made during a search; all symbols that are seen following the order *N* string are excluded from a new search if the algorithm shifts to order *N*-1 (i.e., eliminates repeating symbols in lower orders). This makes sense because if the eliminated symbol were the searched symbol, it could be compressed in step *N*, and the algorithm wouldn't be shifted to order *N*-1. The elimination of repeated symbols increases the probabilities of real candidates for a match, improving the compression ratio. For example, P=2/5 compresses to $-\log_2 (2/5) = 1.32$ bits (actually 2 bits). If one repeating symbol is eliminated, the probability changes to 2/4. The bit count is $-\log_2 (2/4) = 1$ bit.

### 6. VPC3

VPC is the base algorithm used in TCgen (trace-compression generator), an automatically generated trace compressor, which employs "value predictors to bring out and amplify patterns in the traces so that conventional compressors can compress them more effectively" [33]. TCgen is a pre-compression stage for other compression methods. Based on several other hardware-compression proposals, we are convinced that a pre-compression phase can effectively improve the trace-compression ratio and compression speed.

The first method proposed was to use prediction to compress traces. VPC1 compares an eight-byte input with a series of predicted values, using 37 different predictors. If at least one of the predicted values is correct, the algorithm outputs the address only for the correct predictor. If no predictor predicts the input data, a miss flag is output, together with the actual data. VPC2 uses gzip as a second-stage compressor, but it's very slow during decompression. Therefore, the final version, VPC3, does not try to compress the trace; instead, it tries to output a stream that is optimized for a second stage algorithm that compresses it. VPC3 is a lossless, single-pass, fixed-memory algorithm, making it ideal for trace transformations.

VPC3 uses four predictors: a last *n* values predictor, a stride predictor, a finite-context-method predictor, and a differential finite-context-method predictor. These four predictors "have been experimentally determined to result in a good balance between the speed and the compression rate of the algorithm on the load-value traces"[34]. The three traces used during the development of VPC3 are the PC and load values of every executed load instruction, the PC and target of all indirect branch instructions, and the PC and effective addresses of each executed store instruction, all from the SPECcpu2000 benchmark suite [34]. The algorithm tables are defined as having the columns correspond to the type of predictor and the number of lines (L) equal to the quantity of predictions stored for each predictor type. The number of stored predictions (L1=s and L2=t in the following figures) under each type of predictor is limited by the available memory, and has to be the power of two.

In VPC3, the index of each table is encoded using a Huffman code to reduce the index length of the most-used predictions. If more than one predictor is correct, the algorithm chooses that with the smallest index. If no prediction is correct, the output is the index of a predictor that already exists and is the closest to the actual data, together with the difference between the dummy prediction and the actual data. Several other enhancements are made: no repeated value is stored in the tables (tables only store different values), the predictions are organized in a last-seen basis to explore locality principles, and the Huffman codes are biased at the beginning to assign the shortest codes to the predictors with the lowest "learn time" (see Chapter II), based on previous analyses, like the differential finite-context method [34].

We have to consider also that "the compression rate depends not only on how many of the trace entries are predictable but also on which predictor can predict them and when a prediction is made since the length of the Huffman codes is different for different predictors and changes over time" [34]. VPC3 requires 27MB of memory [38].

The four predictors are described below:

a. The last *n* values predictor *LV[**n**]*: predicts the next input data based on the *n* previously seen values. The last *n* previously seen values are stored into a FIFO queue, and when a new data is input, it is compared with all *n* stored values. Experimental analyses show that storing four values is enough for a good prediction, outperforming other configurations [35]. See Figure 31.

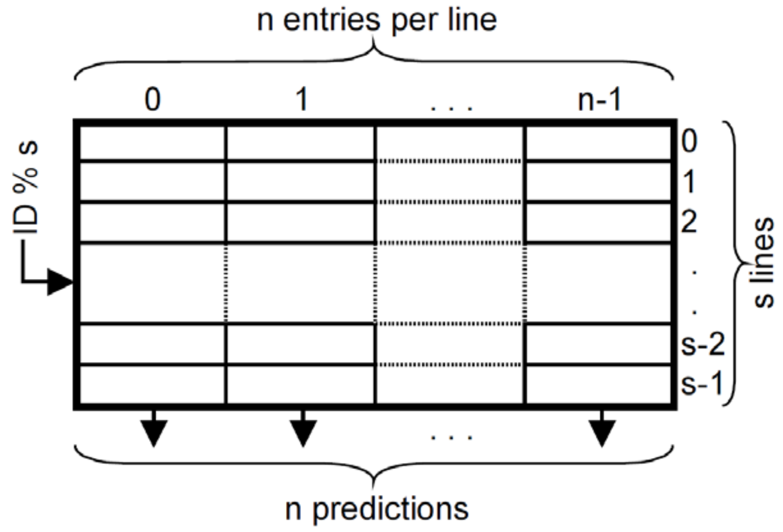


Figure 31. LV[*n*] predictor with s lines (After [39]).

b. Stride predictor *ST[**n**]*: the prediction is performed as before, storing the *n* last-seen values, but instead of storing the data itself, the predictor stores the last-seen value and the next *n* differences from it. To improve performance, a second difference is also stored in a different table, but it's updated only if this value is seen twice [34]. See Figure 32.

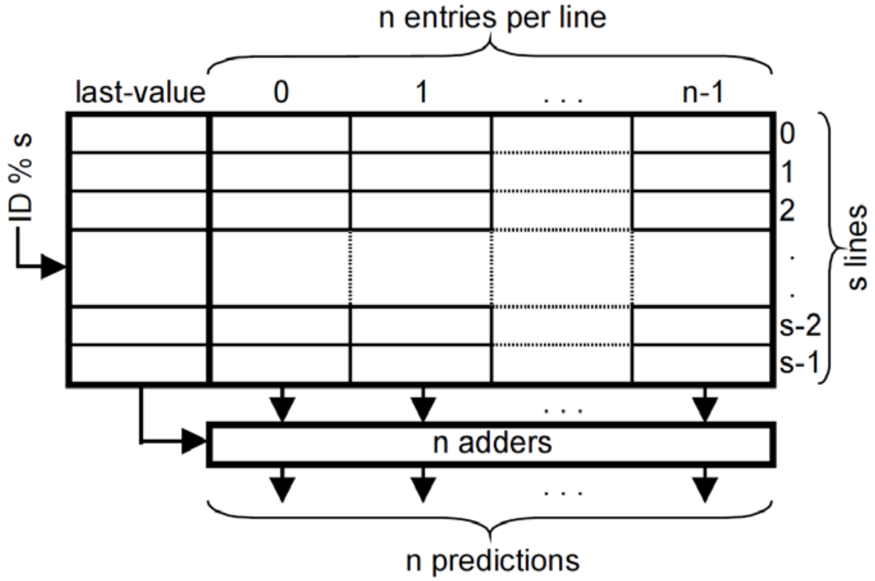


Figure 32. ST[*n*] predictor with *s* lines (After [39]). The first column is the last-seen value, and all others are differences from it.

c. Finite-context-method predictor *FCM**x**[**n**]*: this predictor reads the last *x* symbols seen, stores them in a FIFO sliding window, and then stores the *n* values that follow *x* in a prediction table, using a hash function. Every time a new string is input, it is compared with the *x* values in the queue, and if they match, the next value is predicted as being the one stored in the table addressed by the hash value. The algorithm does not compare the input data with the *x* values in the first table. For space/speed reasons, it just takes the hash of the input value and searches the address pointed to by this hash in the second table; thus, there is no real need for the first table. The values in the second table are only updated if they have made incorrect predictions twice after a good prediction, requiring an additional structure. See Figure 33.

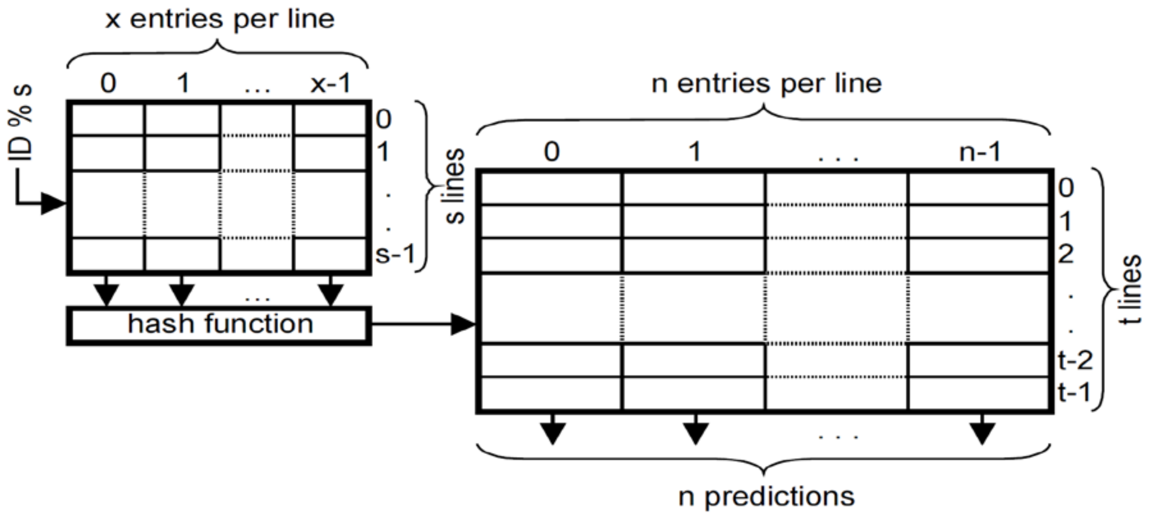


Figure 33. FCMx[*n*] predictor with L1 = s and L2 = t (After [39]).

d. Differential finite-context-method-based predictor *DFCM**x[n]***: this method is the same as the previous method, except that the stored predictors are differences from the last seen one. Using differences instead of values can improve the prediction accuracy by as much as 33% [36]. The hash function is responsible for part of the performance of this method, and any change in the hash function can improve its performance [37]. See Figure 34.

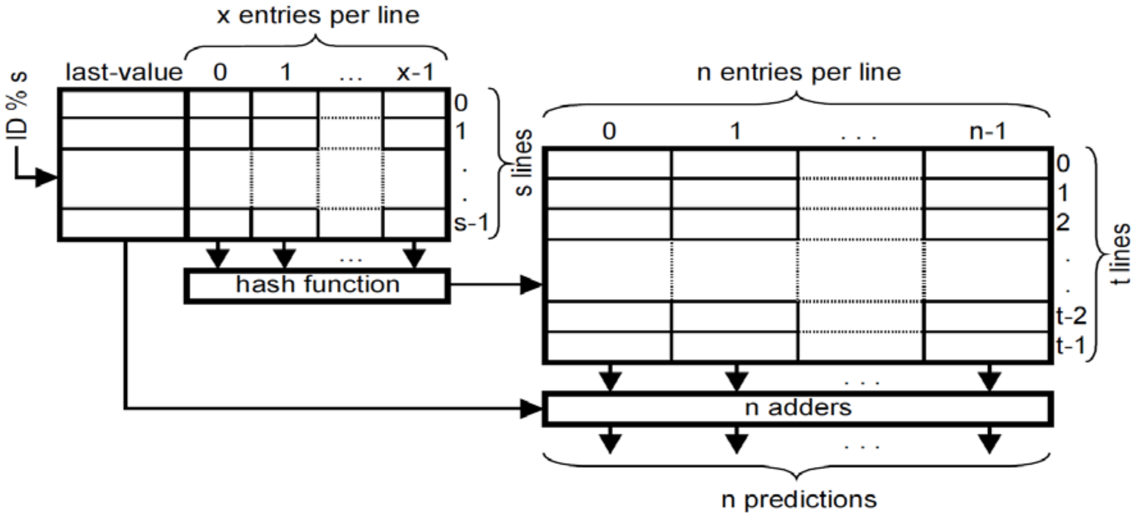


Figure 34. DFCMx[*n*] predictor with L1 = s and L2 = t (After [39]).

# 7. TCgen

> TCgen, is a tool that automatically generates portable, customized, high-performance trace compressors. All the user has to do is provide a description of the trace format and select one or more predictors to compress the fields in the trace records. TCgen translates this specification into C source code and optimizes it for the specified trace format and predictor algorithms[38].

TCgen uses VPC3 as its base algorithm; it transforms traces into highly compressible streams that are sent to a general-purpose compressor. The input data is divided into fields according to the trace specification; a field can be the program counter, the memory address, the instruction counter, etc. Each field is input and compared with a set of predictors; if at least one predictor is correct, the address of this predictor is output to a stream. Otherwise, if no predictor is correct, a miss flag is output to the same stream, and the original data is output to a different stream.

At the end of this pre-compression phase, we have the trace file divided into two streams: a pre-compressed stream, composed of the address of good predictors plus the miss flags for the non-predicted values, and an uncompressed stream, composed of the miss-predicted values, which is searched during decompression when the miss flag appears in the pre-compressed stream. Both streams are further compressed with a general-purpose compression algorithm like GZIP.

The predictors are stored in tables addressed by an index, which is a modular function of a predefined field (ID field) and the number of lines in the table (*ID mod s*). The ID is computed from one specific index of one field in the algorithm. This field (pointed to by the user as the ID field) functions as a reference for the other predictors, but does not have a reference for itself. Therefore, the field defined as the ID field must have only one line in its prediction table (L1=1) and must be carefully chosen.

To generate the C code for this process, the user navigates to the following website: http://www.csl.cornell.edu/~burtscher/research/TCgen/ and types the trace specifications: the header length, if there is a header; the lengths of the fields in bits; the types of predictors to be used, and the predictor specifications (the ID field and the compressor to be used on the second phase), all following the grammar below:

*TCgen Trace Specification;*

***size***-*Bit Header;*

***size***-*Bit Field **1** = {L1 = **s**, L2 = **t**: LV[**n**], ST[**n**], FCM**x**[**n**], FCM**x**[**n**]};*
.
***size***-*Bit Field .. = {L1 = **s**, L2 = **t**: LV[**n**], ST[**n**], FCM**x**[**n**], FCM**x**[**n**]};*

*ID = Field ..;  the L1 in this field has to be 1*

*Compressor = '**general purpose compressor**';*
*Decompressor = '**general purpose decompressor**';*

When specifying the traces, the user must provide an e-mail address, and the generated C code is sent to the user, who saves and compiles it. Other usage details can be found in [39].

## G. FURTHER DISCUSSION

### 1. Combining

Dictionary methods compress strings of symbols (helping compression) but ignore the context (hindering compression), while prediction methods have a probability memory (helping compression) but compress one symbol at a time (slowing compression). Therefore, a hybrid method that combines the advantages of both approaches is needed. Such a method exists: dictionaries that use some context and probability when they store, for example, the last-seen value, or search the dictionary by some order, like LZW, LZT or LZMW.

### 2. Data-Compression Patents

Having presented several compression algorithms, and before we proceed to implementations, we discuss patents, since most implementations of these algorithms are proprietary. David Salomon says: "It is generally agreed that an invention or a process is patentable but a mathematical concept, calculation, or proof is not. An algorithm seems to be an abstract mathematical concept that should not be patentable. However, once the algorithm is implemented in software (or in firmware) it may not be possible to separate the algorithm from its implementation. Once the implementation is used in a new product (i.e., an invention), that product—including the implementation (software or firmware) and the algorithm behind it—may be patentable [9]. ”

From the presented algorithms, GZIP is “free from patent claims, is faster, and provides superior compression [9],” is a good method for future implementations, and is used by AHA (AHA products group of Comtech EF Data Corporation) in its data-compression hardware.

### 3. Trace Compression

The primary motivating application of our proposed 3D data-ransformation processor is to collect, compress, encrypt, and transmit traces for analysis of program behavior for processor design and security research. The kind of trace to collect is a crucial parameter of the design because it will guide the choice of algorithm: the trace has to be complete enough to represent the processor’s behavior and concise enough to allow good compression. Jones and Zorn present guidelines for the format of traces [43]:

1. Expressiveness: “The trace format must be able to express enough information that the resulting traces can be used to make research contributions in the field” [43].

2. Compactness: “Compact encodings allow larger, more representative traces to be created and shared” [43].

3. Flexibility: “Perhaps the most important goal is to design with the understanding that additional information (and potentially entirely new formats) will be needed” [43].

Traces can record different information depending on the objectives of their collection and analysis. “For example, control flow analysis needs only a trace of executed basic blocks or paths. Cache studies require address traces, and more complex processor simulations need instruction words as well. Branch predictors can be evaluated using traces with only branch-relevant information, such as branch and target addresses, and branch outcome, and ALU unit simulations require operand values. For example, the Dinero trace format record consists of the address of memory reference and the reference type—read, write, or instruction fetch, and BYU traces also include additional information, such as the size of the data transfer, processor ID, etc.”[32]. ATUM traces (address tracing using microcode) also include the process ID and encompass information about system activity, such as mapping between physical and virtual memory at each translation look-aside buffer miss [44]. An IBS trace record contains the operation code and the user/kernel indicator [31].

Our proposed system will collect traces using hardware probes (direct links) already built in the computation plane that provide the necessary access for the control plane and allocate space for buffers and dictionaries without increasing the area. In the following, we summarize some comments about hardware trace collection from [45].

The first consideration is the size of the buffers at the trace collection point; the sizes must be chosen carefully in order not to stall the compressor or slow it down. "If a long, continuous address trace is desired, then the buffer must either be very large or there must be some way to stall the host whenever the buffer becomes full. It is usually only possible to stall the processor," which is not desirable in our case [45]. "If there is no way to stall the system, then several discontinuous address-trace samples can be acquired and concatenated together. In either case, the resulting trace exhibits a form of distortion that we call *trace discontinuity*" [45]. Special hardware has been built to avoid stalling the processor. Biomation Corporation built a trace-collection system in 1983 with 80 million trace-buffer entries. One way to reduce the size of the traces is to ignore some primary caches and TLB behavior; although they are important, "a trace of just cache misses is by no means worthless, (…) such a trace can still be used to simulate other cache configurations, albeit subject to certain restrictions" [45]. Those are approaches that reduce the efficiency of our architecture, and are not desirable. If the trace buffer is sufficiently large, we can capture complete sequences with "both user and kernel memory references, and free of most forms of trace distortion" [45]. Since some Intel processors have more than 10 MB cache, similar memory can be implemented in the same area on the control plane, allowing more than 130,000 entries of 64 bits, a figure that can be used as the size of the trace-compressor buffers.

Another important consideration is that the collected traces are difficult to interpret. "Hardware events such as cache misses, integer and floating-point-unit stalls, exceptions and interrupts all must be separated from run cycles to determine the actual type (read, write, execute) and size (word, half word, byte) of the memory references made by a monitored processor" [45]. We may also provide some access to the OS data structures to emit markers or other clues to reverse-translate the physical addresses captured by the control plane to their matching virtual addresses, if the analyzer requires it [45].

The number of processors is another issue. "Parallelism complicates tracing by increasing the volume of data that must be recorded, introducing uncertainty into the ordering of instruction and memory references between processes, and by allowing programs with indeterminacies that are affected by tracing" [46].

The number of processes can also determine the trace formats, "Some systems can trace all processes running on a computer, sometimes even including the operating system.. The values of a multitasking trace depend on its intended application" [46].

Fortunately, traces have similarities and properties that are not fully harnessed by general-purpose compression, but can be exploited by a pre-compression phase. "Combined instruction and data address traces can be compressed by recording only offsets from previous trace records of the same type, by linking data addresses to the corresponding dynamic basic blocks or loops, or by regenerating values using abstract execution or prediction" [29]; we summarize some of these techniques in the following.

## H. 2D COMPRESSION HARDWARE

### 1. Parallel Dictionary LZW Plus Adaptive Huffman [27]

Figure 35 shows the proposed architecture, which uses two-stage compression hardware. In the first stage, the string is encoded using *n* parallel dictionaries, each with a different size word. The store and search processes are performed simultaneously in all dictionaries. If a match is found, the codeword for the match is output with its dictionary address. The dictionary of order one has the symbols of the alphabet preloaded (see LZW description). In the second stage, the output of phase one, which is a fixed-length string consisting of a match/miss codeword along with a data address, is encoded using an adaptive Huffman method. The idea is to use the statistical distribution of address/codeword to turn it into a variable length code using fewer bits. Instead of using a tree structure, it uses an ordered list to save search time and memory space. The order is established by swapping one symbol with its adjacent symbol when it appears; therefore, most symbols encountered traverse the beginning of the list.

The hardware architecture of this technique is implemented as follows:
The four dictionaries are implemented with the use of a 296-bit content addressable memory (CAM): (296 = 64 bits (B) x length 2 dictionary word (w) + 32B x 3w + 8B x 4w + 8B x 5w). A five-bit shift register is used to store the input, and a priority encoder is used to select the longest match.

The adaptive Huffman scheme is implemented using a 414-bit CAM for the priority list. The resulting chip has a 4.3 x 4.3 mm$^2$ area, with a core area of 3.3 x 3.3 mm$^2$, power dissipation between 632 and 700mW, and operating frequency of 100 MHz (limited by the cycle time of the CAM) [27]. The amount of data reduction is about 39.95% in average (a compression ratio of 2.5:1).

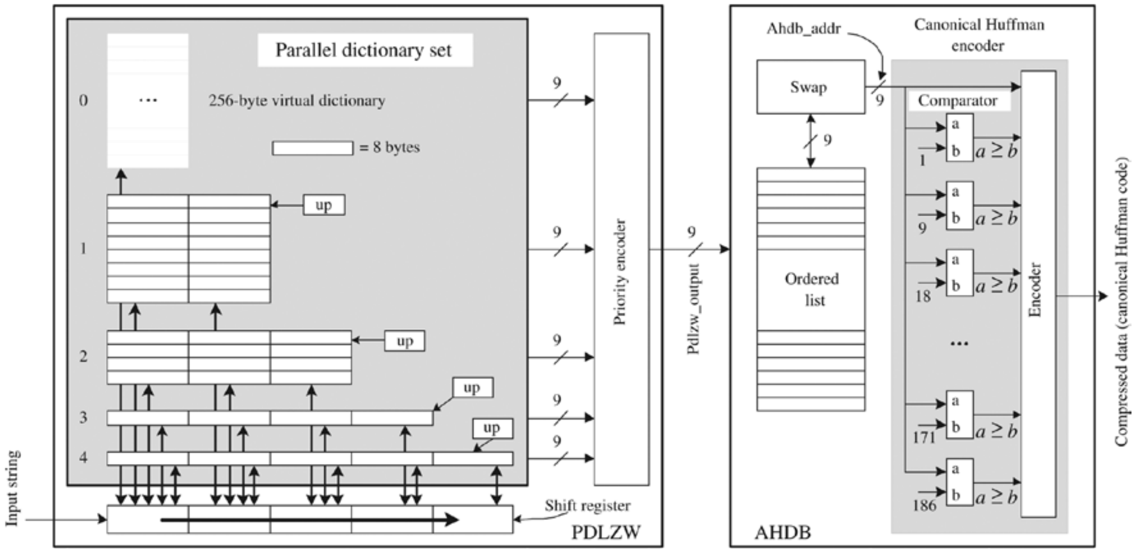


Figure 35. Hardware proposal of Parallel Dictionary LZW plus adaptive Huffman showing the four variable-length dictionaries on the left for word lengths from five to two, and the adaptive Huffman priority queue on right (From [27]).

### 2. X-MatchPRO [28].

This hardware uses a fixed four-byte dictionary word of previously seen symbols and a match or partial match of those symbols, shown in Figure 36. The dictionary is updated using a move-to-front strategy (MTF). When it becomes full, the last symbols are deleted. The token has four fields: a match/miss bit, the matched address, the Huffman-encoded match type (full, partial, run length), and the unmatched literals. The run length is an eight-bit field that encodes the length of a match, from 0 to 255 ($2^8$).

The hardware architecture of this scheme is implemented as follows:
X-MatchPRO uses a 16-bit data register and six registers for command and control; the input data can vary from eight bytes to 32 Kbytes. The dictionary uses a 16 x 4 bytes/word CAM with predefined word lengths of 16, 32 or 64 bytes. Input data is compared with the dictionary data using XOR gates; AND gates are used to select just one output bit per word position. The clock period is reduced with pipeline registers to achieve better throughput. The percentage of data reduction is about 51% to 58% on average (a compression ratio between 1.96:1 and 1.72:1).

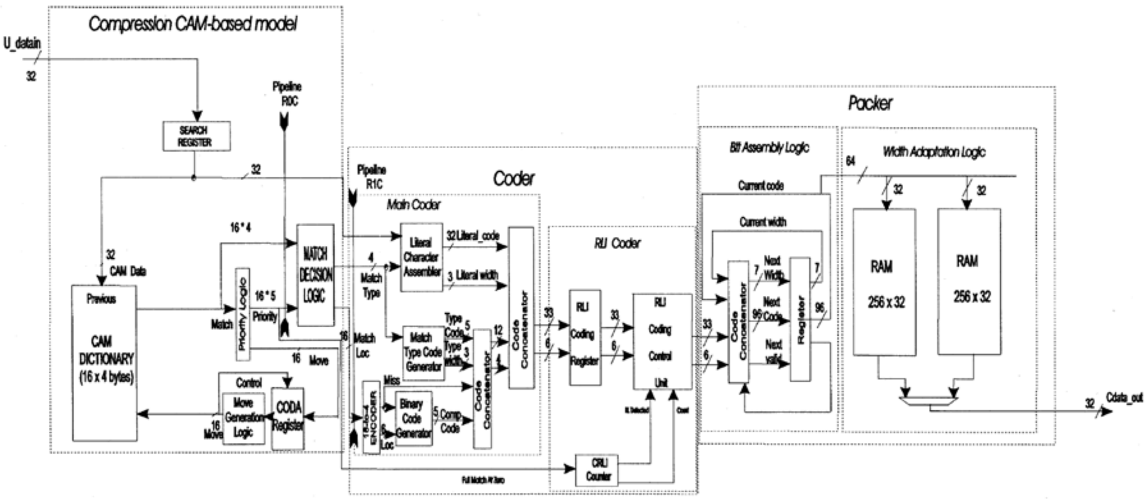


Figure 36. Hardware proposal of X-MatchPRO, showing at the far left the dictionary CAM, and at the center the Huffman coder (From [28]).

### 3. Branch-Predictor Compression Plus Variable-Length Code [29]

The focus of this scheme is to compress execution traces so that a program's execution path to be recreated for debugging analysis by recording the program counter (PC) when changes in the program flow occur, and recording one of the following:

- The branch target address (BTA), direct or indirect, in case of a control-flow instruction; or
- The exception-handler target address (ETA), in case of an exception.

PC values are substituted by the sequential counting (SC) of instructions executed since the last change in the control flow [29].

The key idea is to implement branch-prediction hardware on the host machine, and software for debugging analysis on the analyst machine, as in Figure 37. The host outputs data (indirect/direct BTA or ETA) only when a miss-prediction event occurs, together with a counter (SC) from the previous miss prediction. If an indirect BTA is the miss-predicted event, it also outputs the correct target address (TA); during the debugging process the software will follow the same original execution and will count down starting from the miss-prediction counter (SL) received from the hardware. When the countdown reaches zero, the software knows that the actual prediction is wrong, and will take the opposite branch (opposite BTA) for a direct BTA, the received TA for an indirect BTA, or handle the exception (ETA). The output trace is also coded using a variable-length code.

The hardware architecture of this proposal is implemented as follows:

The compression hardware is coupled with the CPU from which it takes the primary information such as the BTM, ETA, PC, instruction type, and exceptions. The TA output is the difference between the TA and the previous TA output using a simple XOR scheme. The compression ratio achieved is 1:419 (using 2,800 logic gates).

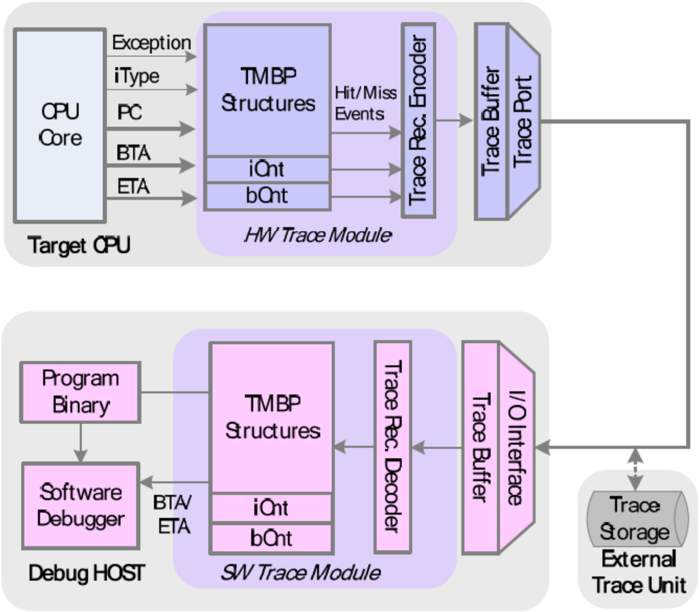


Figure 37. Branch-predictor compression plus variable length code (From [29]). This figure shows the host machine with the hardware model at the top, and the software in the analysis machine at the bottom.

### 4. Stream-Based Compression (SBC) [30], [31]

For this scheme the authors base their algorithm on the fact that “most programs generate only a small number of unique instruction streams…. The starting address (SA) and length (SL) uniquely identify an instruction stream” [30], and instruction addresses often have a regular stride [30], [31] show in Figure 38. They give as an example the fact that “the average instruction stream length is about 12 instructions for the SPEC CPU2000 integer applications and about 117 instructions for the floating-point applications, with a maximum length of 3162 instructions and a minimal length of one instruction”[30], [32]. The compression is divided into instruction-address compression and data-address compression, generating two output files that can be further compressed using a dictionary method.

The stream-based compression method simplifies (compresses) the traces by dividing them. The addresses (SA) are kept in stream caches and stream buffers, and instructions are kept in data-address stride caches (DASC). During a search, if a match occurs, just the pointer to the data is output, as follows [30].

The trace is read, the instruction type and first data address are kept in the stream buffer, and the stream length (SL) is incremented until the next stream is detected. When a new stream is detected, the algorithm stops this phase and performs a search for the old stream in the stream cache (using a hash table to speed up the process). If there is a match (hit), the output is an index to the stream-cache table, which points to the field that contains the data address (SA) and length (L). If no match is found (miss), the algorithm outputs a miss flag, the first data address, and its length, and updates the stream cache. The decompression similarly accepts as input the stream index, and then the corresponding stream table field is accessed, giving the address of the first instruction, and respective stream length.

For the data address, the approach is to read the actual data address and the respective program counter. From the program counter, an index is computed, which is the index into the stride table. This table contains previous data addresses (LDA) and previous strides. The address pointed to by the index is compared with the actual address. If the stride remains the same as the previously stored stride, the algorithm outputs a one-bit hit flag. If the stride changes, the new data address and stride are written to the table, and the miss flag and the actual address are output.

The hardware architecture of this scheme is implemented as follows: the overall size is 7629 bytes. Due to its small size, its speed of operation is the same as the CPU clock frequency [30]. This makes it suitable for a system-on-chip (SoC) type of hardware architecture. The compression ratio achieved is 125.9:1 for instruction-address traces and 6.1:1 for data-address traces.

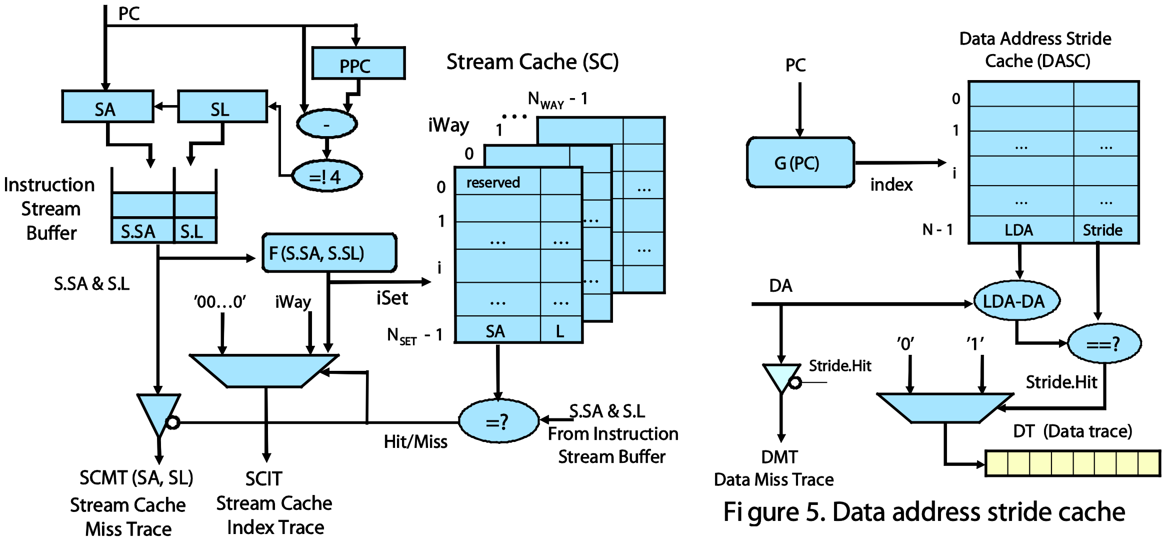


Figure 38. Stream-Based Compression (From [30], [31]). The compression is divided into instruction and data-address compression, generating two output files that can be further compressed using a dictionary method. Instruction address compression (left) is divided into address (SA) and length (SL), which are compared with cached values. If a match occurs, a hit flag is output. In data-address compression (right), input data is compared with cached data. If the strides remain the same, a hit flag is output.

## 5. Reduction, Encoding Plus LZ [47]

This scheme has three phases: the first phase is branch/target filtering, which computes only discontinuous addresses to reduce the trace size; the second encodes the first phase's addresses to reduce the average bit length; and the third phase is a common LZ compression algorithm. This method also has the goals of a real-time compression and compact size compatible with system-on-chip (SoC) implementation.

The trace has sixteen bits for each CPU cycle, five bits of pipeline status information, eight bits of indirect PC, and three bits of breakpoint qualification information. The author argues that "from these addresses, host-side debug software can reconstruct the instruction-execution trace" [47].

The first phase, shown in Figure 39, takes advantage of the sequential instructions presented inside a basic block (a sequence of linearly executed instructions initiated by a target instruction and ended with a branch instruction). The algorithm stores a sequential offset value from a basic block in one register, and every time a new data is input, the algorithm compares the offset with the stored one. If the offset is different, a new branch has occurred. The actual data is a target address for the next basic block, and the previous data is the branch address of the previous basic block. This phase only outputs the target and the branch addresses; all other traces are omitted, as they are reconstructed from the stored offset.

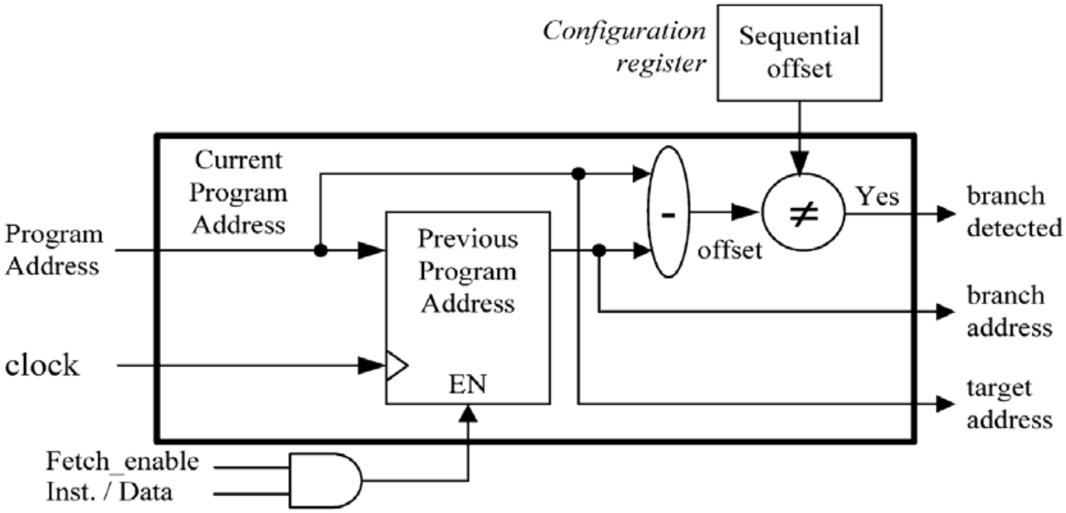


Figure 39. Phase one (From [47]). The hardware inputs all traces and outputs just the target and branch examples.

The second phase receives the target and branch addresses from phase one and encodes it. To encode the target address, there are two mechanisms: the first is a slicing module that divides the sixteen-bit address into small four-bits chunks and appends a 1 between chunks that are part of the same address or a 0 if they belong to different addresses. The second mechanism is a slice encoding that, assuming consecutive target addresses from different basic blocks are similar, takes the chunks from the actual target address and compares them with the same chunks from the previous target address, outputting just the different chunks. All chunks are sent to an output FIFO buffer and to phase three, as shown in Figure 40.

For the branch addresses, this method exploits the property that in the same basic block, the difference between the target address and the branch address in binary has a large number of leading zeros in the most significant bits, followed by a small string of ones and zeros that represent the difference. The algorithm exploits this property by slicing the address into equal chunks and computing the difference chunk-by-chunk, outputting a chunk only if the difference is nonzero. Therefore, the algorithm represents the branch addresses as a difference from the target address in the same basic block and eliminates all leading zeros, reducing the bit sizes of these addresses.

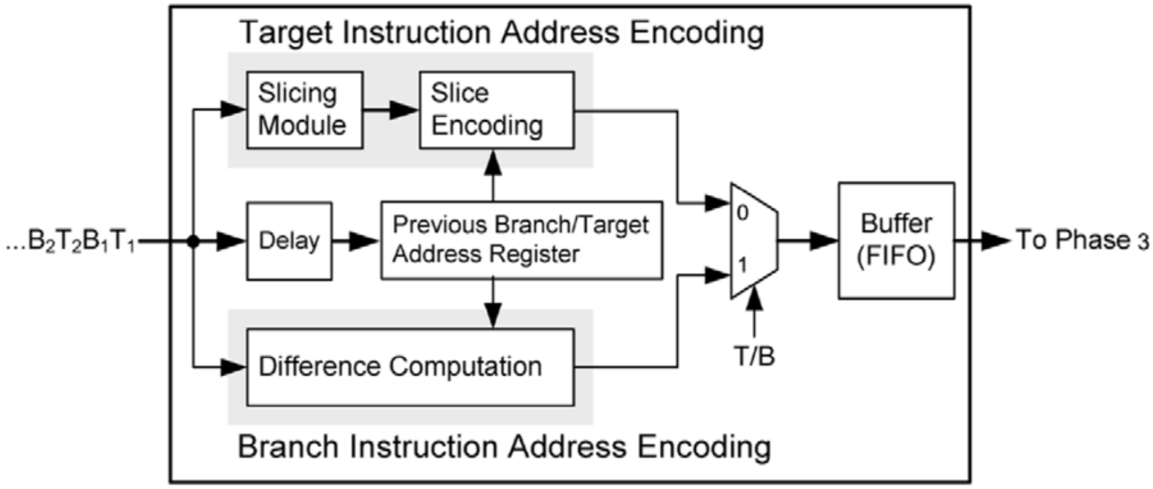

Target Instruction Address Encoding
Slicing Module
Slice Encoding
...B₂T₂B₁T₁
Delay
Previous Branch/Target Address Register
0
1
T/B
Buffer (FIFO)
To Phase 3
Difference Computation
Branch Instruction Address Encoding

Figure 40. Phase two (From [47]). The hardware inputs just the target and branch addresses. For the target address it outputs the comparison that differs from the previous target address sliced chunk-by-chunk, and for the branch it outputs the difference from the respective target address.

Phase three is a simple LZ dictionary that receives data from phase two and compares with a dictionary of previously seen data, as explained before. The advantage is that after slicing data into small chunks during phase two, the size of the dictionary in phase three is much smaller, saving hardware cost and area.

The proposed hardware was implemented in Verilog HDL RTL code, and the compression ratios achieved are 1:3.3 for phase one alone, 1:7.2 for phase one plus two, and 1:454.5 for all three phases combined.

### 6. IBM/AHA [40][41]

As an example of a commercially available 2D compression coprocessor, we describe the IBM/AHA products that use ALDC, an IBM compression algorithm. We highlight it because AHA, a group from Comtech EF Data Corporation, a subsidiary of Comtech Telecommunications Corporation, is the “recognized global leader in satellite bandwidth efficiency and link optimization” [42].

The basic blocks are the microprocessor interface, the input/output interfaces, and the compression/decompression engine, as shown in Figure 41. The microprocessor interface receives the control signals and provides status information to the microprocessor, via externally accessible registers. The input/output interfaces have sixteen-byte FIFO buffers, and the coprocessor receives a single clock input. From the clock reference, it will generate all internally needed clock signals. The hardware can also interrupt data input/output when an almost-full buffer flag is raised, meaning that the device needs more time to transform data.

The hardware architecture of this scheme is implemented as follows:

The hardware is a 28 x 28 x 3.8 mm device with 144 pins. It receives a 5 (+- .25V) VDC power supply and a single clock signal. It is implemented in 0.8 micron CMOS, packed flat in plastic with a molded heat sink. The compression speed is up to 40 MB/s. Other variants can achieve 80 MB/s.

A 512-byte CAM is used for compression to maintain the history buffer. During decompression, a 512-byte RAM is used for the same purpose, and each status register is two bytes wide.

To start compression, the microprocessor sends the appropriate control signal to the microprocessor interface, and the signal is then forwarded to the appropriate register, which is read and decoded. The first data input is the size of the file to be compressed. Compression starts, data is output as it becomes available, and compression ends when the data counter reaches the size of the input file. This counter is a 32-bit register, allowing a maximum of four gigabytes of data to be transferred. Therefore, this is a limitation, as data has to be segmented into four gigabyte chunks. The reliability average failure rate is less than or equal to 100 PPM. The compression ratio achieved is 3:1.

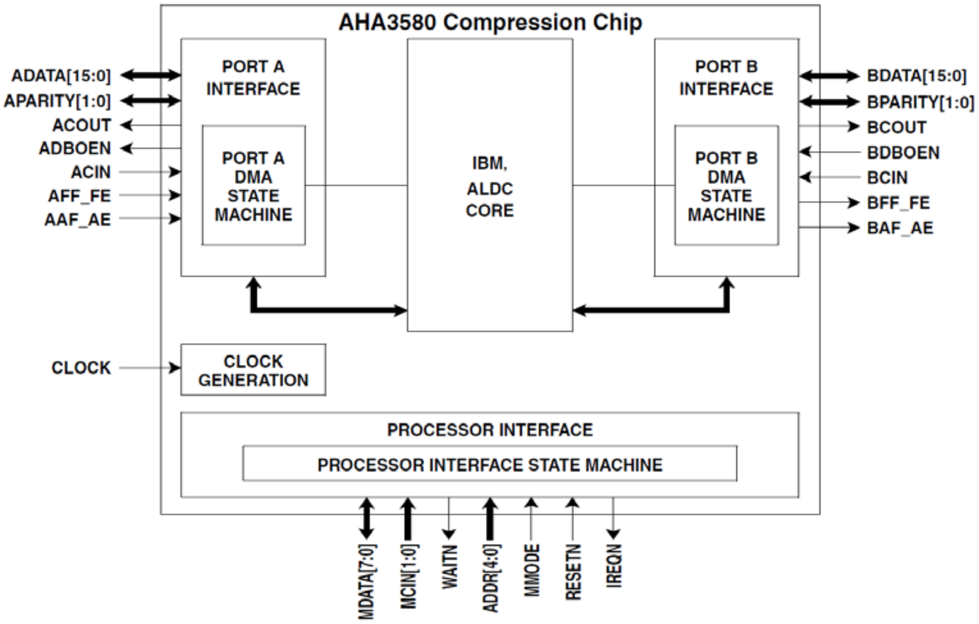


Figure 41. The AHA-ALDC hardware is implemented using a series of 2MB registers in the processor interface, two sixteen-byte FIFO buffers in the interfaces to ports A and B, a 512-byte CAM for the ALDC compression dictionary and a 512-byte RAM for the ALDC decompression dictionary. The device is 28 x 28 x 3.8 mm, and the coprocessor achieves a compression ratio of 3:1 and a speed of 40 MB/s (From [41]).

## I. USAGE SCENARIOS

We present two scenarios in which our proposed 3DIC system can be applied and a third general-application scenario that requires modifications from our original proposed design.

Scenario one: Real-time analysis of malicious behavior. The ability to monitor computation planes fabricated in untrusted foundries by collecting, compressing, and encrypting traces using a low-cost, efficient control plane fabricated in a trusted foundry, providing significant benefits to trustworthy system design. The device sends the compressed and encrypted trace over the network to an analyzer that can automatically recognize suspicious behavior and alert the analyst. The large investment in the development of this architecture is amortized across many users, including ordinary customers who do not require a control plane to be attached and customers with high trustworthiness requirements who will purchase a device with a control plane attached.

Scenario two: Real-time analysis and debugging of software. As explained above, traces are helpful in tracking program execution behavior and determining where and when problems occur. By collecting and compressing those traces in real time, an analyst can speed up the analysis process. Also, by implementing a general-purpose processor in the computation plane, the cost of development of this 3D architecture can be amortized across many customers.

Although we are just discussing trace compression in this thesis, our proposed architecture is not restricted to this domain. Our proposed architecture can also be used for other scenarios, such as satellite compression of hyper-spectral data. Normal images consist of pixels, which are normally 24 bits long, to represent 16.78 million colors. Although this seems sufficient, "There is a large (and growing) field of applications that require images where each pixel is represented by hundreds or even thousands of bits. Such a large set of data is no longer referred to as an image, but is termed *hyperspectral data*" [9].

Military satellites do not rely on image sensors only: enemy equipment can be hidden below some vegetation coverage. Therefore, a satellite "has to measure the radiation reflected from each point on the ground in many wavelengths" [9]. "A typical spy camera consists of a set of sensitive sensors that can measure and record radiation in perhaps 250 frequency bands," much more than the visible frequency boundary, "thus, each 'pixel' in the image taken by such a camera consists of 250 numbers, each an integer of at least 16 bits" [9].

For example, "the AVIRIS sensor (airborne visible/infrared imaging spectrometer) consists of three sensors of 64 frequency bands each plus a fourth sensor with 32 bands, for a total of 224 bands" [9]. Other examples include radar, radar altimeters (both microwaves and laser), radiometers, photometers, sonar (which is a hyper-spectral application because different sound frequencies penetrate the water and are reflected from objects in different ways [9]), medical imaging, and much more [9]. All of these applications need to compress data and send it for analysis, processing, and presentation as quickly as possible, and 3D compression/crypto hardware can provide the solution by reducing the data size by placing the data transformation engine close to where the data is generated, reducing the overall cost, size, and duration of data manipulation.

## J. PERFORMANCE NUMBERS

This topic provides figures from the literature for comparing compression algorithms and their hardware implementations. The figures in each table correspond to the same data, and they originate from the literature cited in the references.

| | **COMPRESSION RATIO** | **MEMORY [bytes]** |
|---|---|---|
| **Adap Huffman** | 62.7 | 8K |
| **LZW** | 44.3 | 48K |
| **LZ78** | 39.6 | 350K |
| **Arithmetic** | 36.6 | 32-1400K |
| **LZ77** | 28.4 | 8K |
| **LZSS** | 27.3 | 2K |

Table 2. Performance figures from the literature on some of the presented algorithms: compression ratio and memory required for compressing the same data as a means of comparison (From [26]).

|  | PROCESS [microns] | GATES [K] | SPEED [MHz] | THROUGHPUT [MB/s] | ALGORITHM | COMPRESSION RATIO | AREA [mm] |
|---|---|---|---|---|---|---|---|
| **Filter/Code + LZ** | - | 51 | 185 | 96 | Filter/Code + LZ | 6.36 | - |
| **ALDC1-40S** | 0.8 | 70 | 40 | 40 | ALDC | 3 | 28 x 28 |
| **PDLZW +AHDB** | 0.35 | 130 | 100 | Compr:16.7-125<br>Decompr:25-83 | LZW + Adapt Huffman | 2.5 | - |
| **Hi/FN** | 0.35 | 100 | 80 | 80 | LZS | 2.25 | - |
| **AHA 3521** | 0.5 | - | 40 | 40 | ALDC | 2.25 | - |
| **AHA3580** | - | - | 80 | 80 | ALDC | 2 | 14 x 14 |
| **X-Match PRO** | 0.18 | - | 50 | 200 | X-Match PRO | 1.96 | - |
| **AHA3231** | 0.5 | - | 40 | 40 | LZ78 | 1.92 | - |

Table 3. This table summarizes performance figures from hardware implementations in the published literature (From [28]). The figures are organized in descending order of compression ratio and throughput

|  | COMPRESSION RATIO | MEMORY [bytes] |
|---|---|---|
| **GZIP-9** | 3.05 | - |
| **GZIP-1** | 2.6 | - |
| **LZS** | 2.3 | 8K |
| **ALDC** | 2.1 | 512 |
| **VPC3** | - | 27M |

Table 4. Like Table 3, this table summarizes compression ratios and memory requirements of some of the algorithms presented for compressing the same data as a means of comparison (From [26]).

THIS PAGE INTENTIONALLY LEFT BLANK

# IV. CRYPTOGRAPHY

## A. INTRODUCTION

Cryptographic coprocessors are used in a variety of systems for which the efficiency of encryption and decryption is crucial. Originally developed for military cipher machines [48], cryptographic coprocessors have uses in smart cards, banking, telecommunications, networking, aerospace, and high-assurance computing platforms. A crypto coprocessor is a custom circuit for carrying out cryptographic transformations, often embedded in a tamper-resistant packaging. Systems will often combine a cryptographic coprocessor with a general-purpose processor, key storage, and other elements. A crypto coprocessor may implement just one algorithm or may support a variety of ciphers, e.g., DES, RSA, SHA-1, etc. A successful design and implementation requires careful balancing of tradeoffs between speed, cost, power, and security.

## B. DESCRIPTION OF A CRYPTOGRAPHIC COPROCESSOR

A cryptographic coprocessor is a special-purpose computing environment, many of which environments are designed to withstand various kinds of attacks, e.g., physical probing or side-channel analysis [49], whereas others depend for protection on limiting access to the device. According to Smith and Weingart, crypto coprocessors are “computational devices that can be trusted to execute their software correctly, despite physical attack” [50]. Crypto coprocessors support many applications by implementing various cryptographic operations such as key distribution, key management, the management of digital certificates, encryption mechanisms, and decryption mechanisms.

In general, a cryptographic coprocessor is a hardware device comprising the following parts: (1) a CPU, (2) bootstrap ROM, and (3) secure non-volatile memory [51]. In the majority of cases, this specific hardware device is physically shielded to protect it from tampering and side-channel attacks, and the I/O interface is the only way to access the internal state of the device. The device stores cryptographic keys securely. A crypto coprocessor can contain special-purpose hardware in addition to a general-purpose CPU and memory, e.g., high-speed encryption/decryption circuitry [51].

The IBM 4758 and its successor, the IBM 4764, are two secure coprocessors in the market [52],[53]. Several research and development initiatives occurred prior to the development of the IBM secure coprocessors. Steven Kent's 1980 thesis explored the application of what he named "tamper-resistant modules" (TRMs) to protect software [54]. Steve White and Liam Comerford at IBM Research implemented ABYSS (A Basic Yorktown Security System), an architecture that supports and ensures trusted execution of software [55]. Later, Steve White and his colleagues integrated the ABYSS design into an enhanced system, named Citadel [56], [57]. Some of the Citadel prototypes from IBM became the foundation of the Dyad project, built by Bennet Yee and J. D. Tygar in the early 1990s [58], [59], [60], [61]. Finally, Yee and Tygar developed four types of electronic-commerce applications on top of a secure coprocessor, including copy protection for software, electronic cash, and electronic contracts.

## C. THE HSSEC HIGH-SPEED CRYPTOGRAPHIC COPROCESSOR

In our proposed 3DIC implementation of a data-transformation coprocessor, we take inspiration from the HSSec high-speed cryptographic coprocessor [62].

Kakarountas et al. designed HSSec [62], to implement two hash functions, SHA-1 and SHA-512, and the symmetric block cipher AES-128 [62]. We share HSSec's design goal of minimizing area and we save some room in the control plane for the compression coprocessor. We also follow HSSec's design goal of maximizing throughput, which HSSec achieves using parallelism. Finally, the HSSec system can operate in electronic codebook (ECB) and cipher-block chaining (CBC) modes. The HSSec achieves a throughput of Gbps (AES, SHA-1 and SHA-512) for XILINX's Virtex II FPGA family [62].

We first describe in depth the cryptographic functions of the coprocessor, AES-128, SHA-1, and SHA-512. Next, we present the architecture of the HSSec coprocessor. Then, we present scenarios for utilizing a cryptographic coprocessor. Finally, based on analysis of the literature and other resources on cryptographic algorithms, we study the performance of AES-128, SHA-1, and SHA-512, gathering hard performance figures, including clock rate, throughput, and power consumption.

### 1. SHA-1 algorithm

SHA-1 (Secure Hash Algorithm) is a cryptographic hash function designed by the National Security Agency and published by NIST as a U.S. Federal Information Processing Standard. SHA-1 is the most widely used of the existing SHA hash functions, and it is employed in several widely used security applications and protocols. SHA-1 is an iterated hash function with a 160-bit message digest. According to NIST, it is called secure because it is not computationally feasible to either 1) create a message that can be mapped to a given message digest or 2) create two different messages resulting in the same message digest [64]. Any change applied to a message will produce a totally different message digest. SHA-1 uses word-oriented operations on bit strings, where each word consists of 32 bits [63].

SHA-1 is adequate for hashing a k-bit message, where $0 < k < 2^{64}$. The algorithm consists of two stages: preprocessing; and computation of the hash [64]. The first stage pads the message and divides it into *512*-bit blocks. It next determines the initialization values to be used in the next stage. The second stage generates a message schedule from the padded message and uses this message schedule, together with functions, constants, and word operations, to iteratively produce a series of hash values, one for each round. Each round uses the result of the previous round; SHA-1 requires 80 rounds to generate the 160-bit message digest [62]. The message digest is determined from the final hash value resulting from this iterative process. Appendix A provides a detailed explanation of the entire SHA-1 process.

### 2. SHA-512 algorithm

SHA-512 (Secure Hash Algorithm) is a cryptographic hash function designed by the National Security Agency and published by NIST as a U.S. federal information-processing standard. According to NIST, it is secure because it is computationally infeasible to: (1) find a message that can be mapped to a given message digest or (2) to find two different messages that map to the same message digest. Any change to a message will produce a totally different message digest.

The algorithm consists of two stages: preprocessing; and computation of the hash [64]. The first stage pads the message and then divides it into *m*-bit blocks. It then determines the initialization values needed in the next stage. The hash-computation stage first generates a message schedule from the padded message and it uses this message schedule, together with functions, constants, and word operations, to iteratively generate a series of hash values. The final hash value resulting from this iterative process is the message digest.

SHA-512 is adequate for hashing a k-bit message, where $0 < k < 2^{128}$. The preprocessing phase pads the message and divides it into 1024-bit blocks. These specific blocks are needed to generate the message schedule. SHA-512 uses 80 rounds to generate the 512-bit message digest [62]. Each round requires the result of the previous round. Appendix B provides a detailed explanation of the entire SHA-512 process.

### 3. AES-128 algorithm

AES is a round-based symmetric block cipher. It processes 128-bit data blocks and uses a key whose length can be 128, 192, or 256 bits [66]. Table 5 shows the available combinations of key, round, and block. Our proposed 3DIC incorporates AES-128 circuitry.

| | **Key Length (Nk words)** | **Block Size (Nb words)** | **Number of Rounds (Nr)** |
|---|---|---|---|
| AES-128 | 4 | 4 | 10 |
| AES-192 | 6 | 4 | 12 |
| AES-256 | 8 | 4 | 14 |

Table 5. Key–Block–Round Combinations (From [66]).

#### *a. AES-128 Transformations*

AES-128 belongs to the family of round-based symmetrical block ciphers. AES-128 accepts a 128-bit data block as input and performs many different transformations on this block. During the encryption phase, the input block of the AES is plaintext, and the output block is ciphertext. All the other intermediate values of the block, during its transformation from plaintext to ciphertext, are *states* [63], [67]. A state is a four-by-four matrix of bytes, as depicted in Figure 42.

S

| | | | |
|---|---|---|---|
| $S_{0,0}$ | $S_{0,1}$ | $S_{0,2}$ | $S_{0,3}$ |
| $S_{1,0}$ | $S_{1,1}$ | $S_{1,2}$ | $S_{1,3}$ |
| $S_{2,0}$ | $S_{2,1}$ | $S_{2,2}$ | $S_{2,3}$ |
| $S_{3,0}$ | $S_{3,1}$ | $S_{3,2}$ | $S_{3,3}$ |

Figure 42. An AES state (From [66]). The first byte of the block resides in the upper-left corner of the matrix; remaining bytes fill out the rest of the matrix. The AES algorithm transforms a plaintext block to a ciphertext block. The intermediate values of the block are states, and the final value of the block is the ciphertext.

AES transforms the 128-bit input/output block as follows: the first byte of the block resides in the upper-left corner of the matrix, and the remaining bytes fill out the rest. The AES algorithm transforms a plaintext block to a ciphertext block. The intermediate values of the block are states, and the final value of the block is the ciphertext [63][67]. AES performs four transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey, which are performed in that order repeatedly. Each of these transformations, described below, maps a 128-bit input state to a 128-bit output state [63]. A round of encryption applies each transformation once. For AES-128, ten rounds must be performed, as shown in Table 5. The following describes the four transformations in detail:

- The SubBytes transformation is “a nonlinear byte substitution that operates independently on each byte of the state using a substitution table (S-box)” [66]. Each byte of the input state is replaced according to a substitution table (S-box)

[63]. The S-Box computes the multiplicative inverse in the Galois Field $GF(2^8)$ with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ followed by an affine transformation (one transformation consisting of a multiplication by a matrix followed by the addition of a vector) [63][66]. Figure 43 illustrates the effect of the SubBytes transformation on the State. This transformation provides resistance to differential and linear cryptanalysis attacks [65].

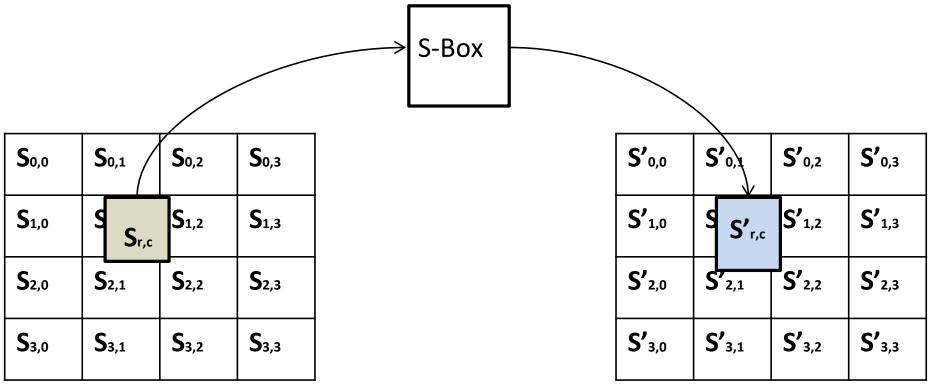


Figure 43. SubBytes() applies the S-box to each byte of the state (From [66]). Each byte of the input state is replaced using the same substitution table (S-box).

- In the ShiftRows() transformation, the bytes in the last three rows of the state are cyclically shifted over different numbers of bytes according to offset values [66][67]. For instance, row one contains elements **$S_{1,0}$ - $S_{1,1}$ - $S_{1,2}$ - $S_{1,3}$;** after the ShiftRows() transformation, row one is rotated by one position to the left. The first row, *r* = 0, is not shifted. Figure 44 illustrates the ShiftRows() transformation [66]. This transformation causes diffusion of the bits over multiple rounds [65].

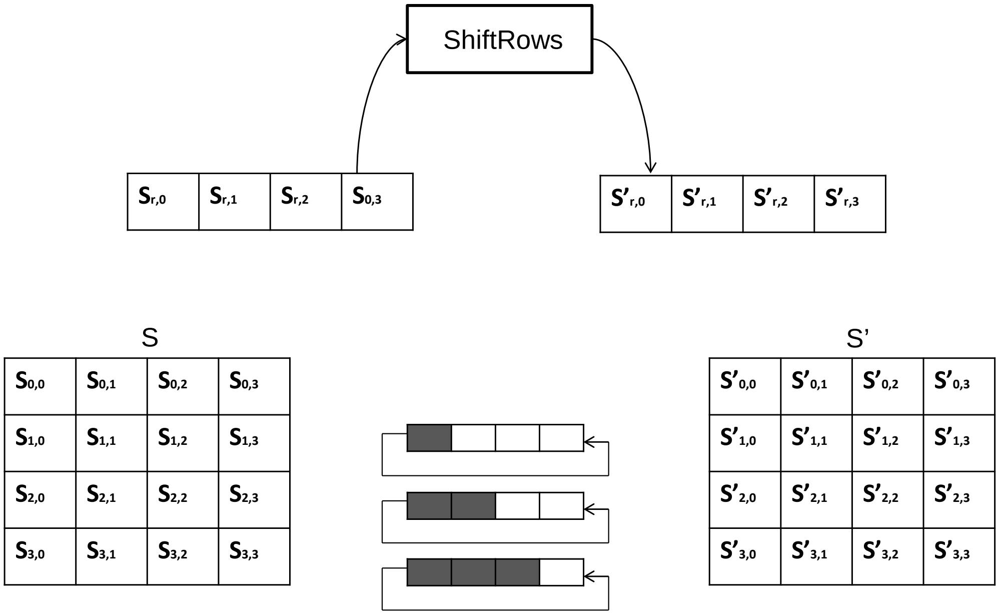


Figure 44. ShiftRows() cyclically shifts the last three rows in the state (From [66]). For instance, row one contains elements **S$_{1,0}$ - S$_{1,1}$ - S$_{1,2}$ - S$_{1,3}$;** after the ShiftRows() transformation, row one is rotated by one position to the left. The first row, *r* = 0, is not shifted.

- The MixColumns() transformation, according to NIST, “operates on the state column-by-column, treating each column as a four-term polynomial” over Galois field $GF(2^8)$, and it is multiplied modulo $x^4+1$ with the constant polynomial, $a(x)=\{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$ [65], [66]. This transformation maps between a column of the input state and a column of the output state [63]. Figure 45 illustrates the MixColumns() transformation.

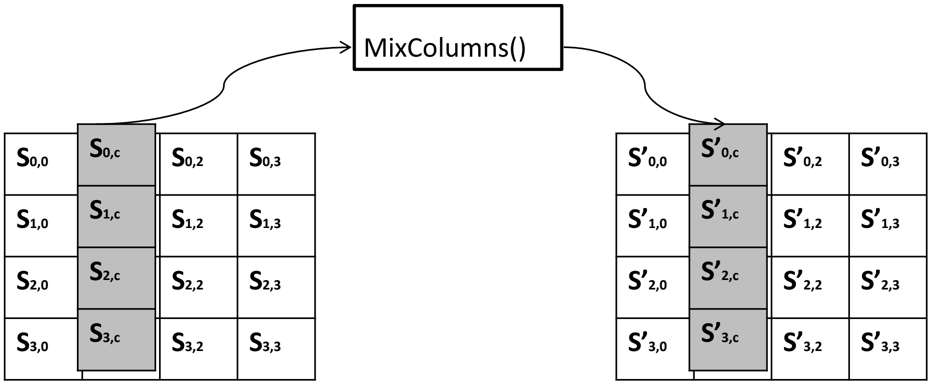

MixColumns()
$S_{0,0}$ $S_{0,c}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,c}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,c}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,c}$ $S_{3,2}$ $S_{3,3}$
$S'_{0,0}$ $S'_{0,c}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,c}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{2,0}$ $S'_{2,c}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,c}$ $S'_{3,2}$ $S'_{3,3}$

Figure 45. MixColumns() operates on the state column-by-column (From [66]). This transformation maps between columns of the input and output states.

- AddRoundKey: According to NIST in this transformation, "a round key is added to the state by a simple bitwise XOR operation." The AddRoundKey transformation is self-inverting. It maps a 128-bit input state to a 128-bit output state by performing an *"xor"* operation on the input state with a 128-bit round key [63], [66].

The four transformations described above are applied to a 128-bit input block in sequence in order to perform AES encryption or decryption. For both encryption and decryption, the transformations are grouped into rounds. There are three different types of rounds: the initial, the normal, and the final [63]. The transformations and sequence of the rounds are depicted in Fig. 46. The number of rounds, *N*r, depends on the key size. AES -128 consists of the following basic steps [63]:

1. For a specific plaintext (κ), initialize state to be (κ), and execute AddRoundKey, which performs an "*xor*" of the RoundKey with the state.
2. For each of the first nine rounds, perform SubBytes on the state using a substitution table (S-box). Then, perform ShiftRows on the state, followed by MixColumns, and AddRoundKey.
3. Perform SubBytes, ShiftRows, and AddRoundKey.
4. The ciphertext (λ) is the state.

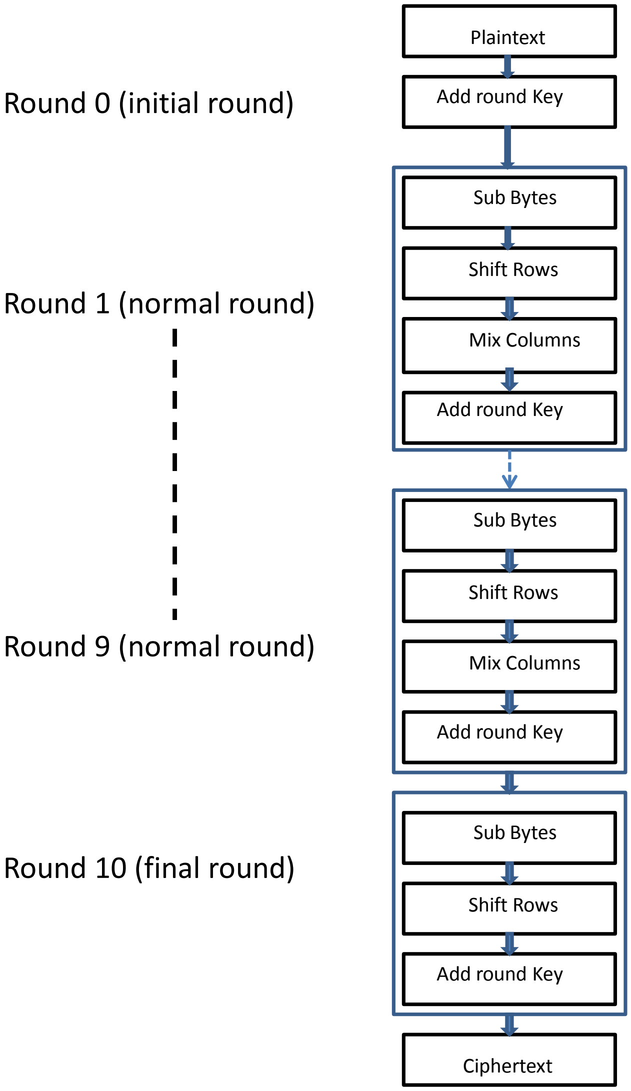


Figure 46. The operation of the AES-128 algorithm, where *N*r = 10 for cipher keys of length 128bits (From [65], [68]). Each round uses a round key derived from the original key (the round-zero key). Each round starts with an input of 128 bits and produces an output of 128 bits. First, it performs AddRoundKey, using the original key (the round-zero key). Next, for each of the first nine rounds, it performs SubBytes on the state using a substitution table (S-box). Then, it performs ShiftRows on state, followed by MixColumns and AddRoundKey. Finally, during the tenth round it performs SubBytes, ShiftRows, and AddRoundKey using the tenth round key. The ciphertext is the 128-bit output block.

### *b. AES-128 Key Expansion Process*

The initial 128-bit cipher key [63] is expanded to eleven 128-bit round keys. The first round key is the cipher key ($\text{RoundKey}_0$), and all subsequent round keys are the result of applying a function to the previous round key. If this function is f, then the process may be modeled as,

$$\text{RoundKey}_i = \text{f}(\text{RoundKey}_{i-1}), \text{for all } 0 < \text{i} < 11$$

The AES-128 algorithm [66] takes the cipher key, K, and performs a key-expansion routine to generate a key schedule. The key expansion generates a total of *N*b (*N*r + 1) words, where *N*b is the block size, and *N*r is the number of rounds. If we use the values in Table 5, we have a total of 44 words. AES-128 needs an initial set of ten words, and each of the four rounds requires ten words of key data. The derived key schedule consists of a linear array of four-byte words, denoted as [wi], with i belong in the range 0 <= i < 44. For the key-expansion process, two functions are required, SubWord() and RotWord(). According to NIST, “SubWord() is a function that takes a four-byte input word and applies the S-Box to each of the four bytes to produce an output word. The function RotWord() takes a word [*a*0,*a*1,*a*2,*a*3] as input, performs a cyclic permutation, and returns the word [*a*1,*a*2,*a*3,*a*0]” [66]. Figure 47 shows how the word array W [0….43] is mapped to the corresponding eleven round keys.

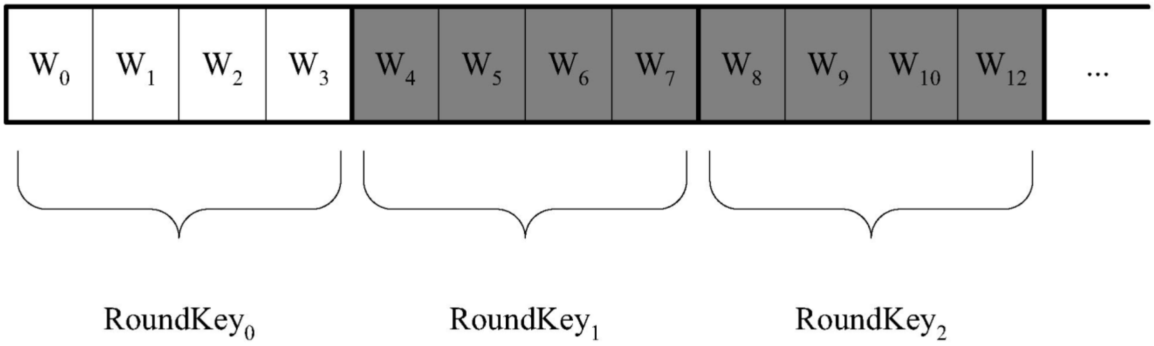


Figure 47. Mapping of the key words to round keys (From [69]).

### *c. Electronic Codebook (ECB) and Cipher Block Chaining (CBC) Modes*

AES-128 supports five modes, including Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes, which provide confidentiality [70]. For our proposed 3DIC cryptographic coprocessor design, we have chosen the symmetrical block cipher AES-128 ECB and CBC modes.

Electronic-codebook (ECB) mode partitions the message into several *n*-bit blocks, adding padding if required, and enciphering each block [71]. A major advantage of ECB is the ability to decrypt the blocks independently, in parallel. Also, if an error occurs during transmission or encryption, the error is isolated and local, affecting only the block in which the error occurred. However, that property can also be disadvantage. If the same key enciphers more than one block, a cryptanalyst can decipher the message using less effort than a brute-force attack. The electronic codebook (ECB) mode is defined as follows [70]:

$$\text{ECB Encryption: } \mathrm{C}j = CIPH_K(Pj) \quad \text{for } j = 1 \ldots n.$$
$$\text{ECB Decryption}: Pj = CIPH_K^{-1}(Cj) \quad \text{for } j = 1 \ldots n.$$

Where $\mathrm{C}j$ corresponds to the the jth ciphertext block, $Pj$ to the jth plaintext block, $CIPH_K(Pj)$ to the forward cipher function of the block cipher algorithm under the key *K* applied to the data block $Pj$, and $CIPH_K^{-1}(Cj)$ to the inverse cipher function of the block cipher algorithm under the key *K* applied to the data block $Cj$."

According to NIST, "in ECB encryption, the forward-cipher function is applied directly and independently to each block of the plaintext. The resulting sequence of output blocks is the ciphertext. In ECB decryption, the inverse-cipher function is applied directly and independently to each block of the ciphertext" [70]. The resulting sequence of output blocks is the plaintext. ECB mode is depicted in Figure 48.

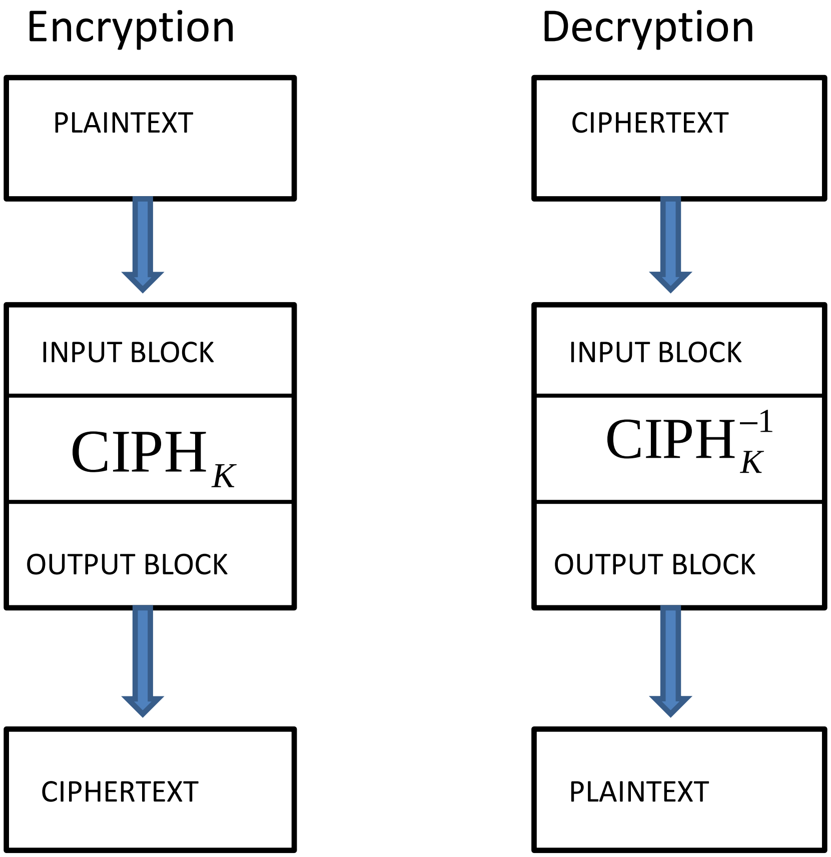


Figure 48. ECB mode operation (From [70]). For a given sequence of plaintext/ciphertext blocks, each block is encrypted/decrypted with the same key, resulting in a string of ciphertext/plaintext blocks.

Cipher-block chaining (CBC) mode [70], [71] computes the XOR of the first plaintext block with an initialization vector (*IV)*, which must be unpredictable. Additionally, the integrity of the IV should be protected, although it does not necessarily have to be secret. The *j*-th plaintext block is XORed with the ($j-1$)-th ciphertext block prior to being encrypted with the block cipher [71]. More formally, the CBC mode is defined as follows: “

CBC Encryption: $C_1 = CIPH_K(P_1 \oplus IV)$

$C_j = CIPH_K(P_j \oplus C_{j-1})$ for $j = 2 \ldots n.$

CBC Decryption: $\mathrm{P}_1 = CIPH_K^{-1}(C_1) \oplus IV$

$\mathrm{P}_j = CIPH_K^{-1}(C_j) \oplus C_{j-1}$ for $j = 2 \ldots n.$

Where $\mathrm{C}j$ corresponds to the the *j* th ciphertext block, $Pj$ to the *j*th plaintext block, IV to the initializing vector, $CIPH_K(X)$ to the forward cipher function of the block cipher algorithm under the key *K* applied to the data block *X*, $CIPH_K^{-1}(X)$ to the inverse cipher function of the block cipher algorithm under the key *K* applied to the data block *X*" [70].

In CBC encryption [70], the first input block is computed by taking the "xor" of the first plaintext block and the IV. The result of this XOR operation is the input to the block cipher, which produces the first block of the ciphertext. Next, the XOR of the first ciphertext block, and the second plaintext block is enciphered to produce the second paintext block.

In CBC decryption [70], to obtain any plaintext block (aside from the first), the inverse cipher function is applied to the corresponding ciphertext block, and the result is XORed with the previous ciphertext block.

Figure 49 depicts the operation of the CBC mode.

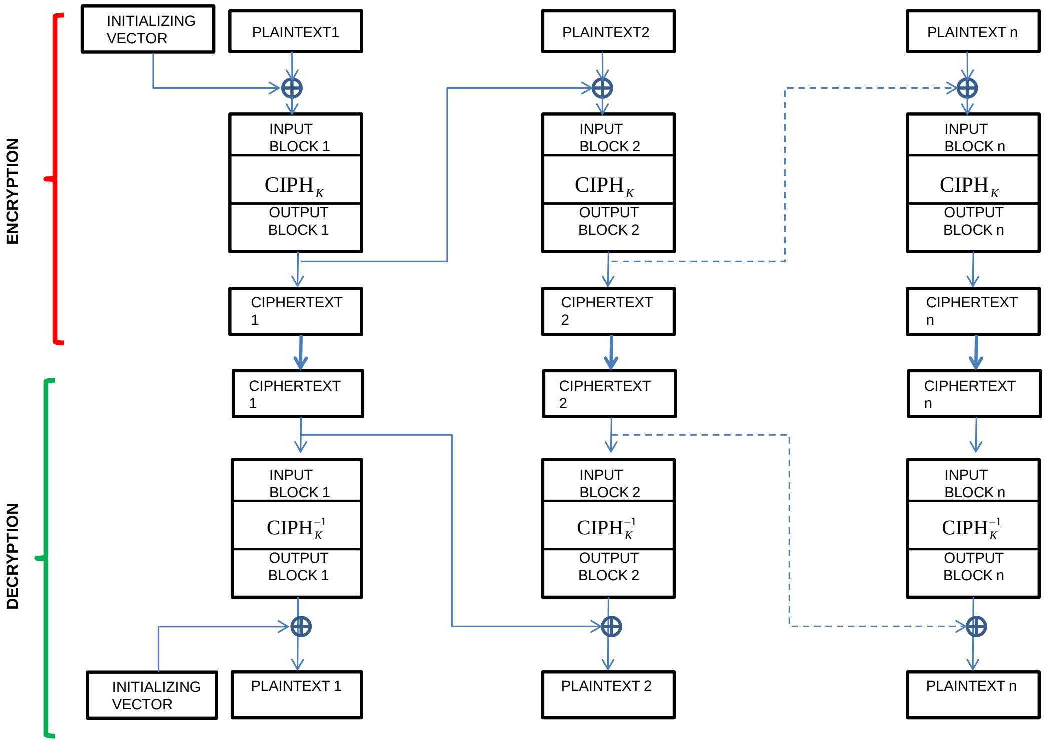


Figure 49. CBC mode (From [70]). In CBC encryption, the first input block is formed by taking the "xor" of the initial block of the plaintext with the IV. The result of the XOR is enciphered, resulting in the first block of the ciphertext. In CBC decryption, in order to decrypt any plaintext block (aside from the first), the inverse cipher function is applied to the corresponding ciphertext block, and the resulting block is XORed with the previous ciphertext block.

## 4. The HSSec High-Speed Cryptographic Coprocessor Architecture

The architecture of the HSSec cryptographic coprocessor [62] consists of a central control unit and various processing elements that are peripheral to the central control unit. For data input/output, a single 32-bit wide data bus is used. Also, for SHA-1 and SHA-512 output, another 32-bit wide data bus is used. Table 2 shows the control signals. The main purpose of the control unit is to coordinate data processing. It is also responsible for the communication between the cryptographic coprocessor and the outside world. The cryptographic primitives (AES-128, SHA-1, SHA-512) are arranged in a parallel orientation and utilize a common 64-bit global data bus. This bus is also responsible for providing data to the memory unit and to the key scheduler.

The key-scheduler block [62] is fundamental to the proper execution and operation of the three cryptographic algorithms supported by the coprocessor. More specifically, it supports two core processes for the cryptographic algorithms. First, it controls the key expansion of AES-128 and generates the message schedules. As described earlier, during AES-128 encryption, the 128-bit cipher key must first be expanded to eleven 12-bit round keys. The first round key is the cipher key ($\mathrm{RoundKey}_0$), and all subsequent round keys are generated by applying a function to the previous round key. If *f* is this function, then the process may be modeled as:

$$\mathrm{RoundKey}_i = \mathrm{f}(\mathrm{RoundKey}_{i-1}), \text{for all } 0 \ < \mathrm{i} < \ 11$$

Therefore, the key scheduler has two basic transformation functions, RotWord and SubWord, described above. The key scheduler also provides the constants required for the hash functions. SHA-1 uses a sequence of eighty constant, 32-bit words, while SHA-512 uses the a sequence of eighty constant, 64-bit words.

The mode interface [62] modifies input to the three cryptographic modules of the cryptographic coprocessor.

The memory block [62] consists of three main parts. The first part is a set of registers used to store the initialization values required by the three cryptographic algorithms. The second part is a general-purpose register file used for storing temporary values for quick access. The last part is the padding unit used for storing fetched data. Specifically, this part consists of eight 128-bit banks. Thus, each bank can support the minimum data-input size required by AES-128. For as for SHA-1, the required number of bits for the input data is 512; thus, four banks are needed. However, the required bits for the input data for SHA-512 are 1024; thus, eight banks are sufficient. Therefore, using eight banks, 128 bits each, for the padding unit meets the requirements for the three cryptographic algorithms.

Figure 50 depicts the architecture of the HSSec cryptographic coprocessor, and Figure 51 shows the memory organization. Table 6 presents the control signals of the coprocessor.

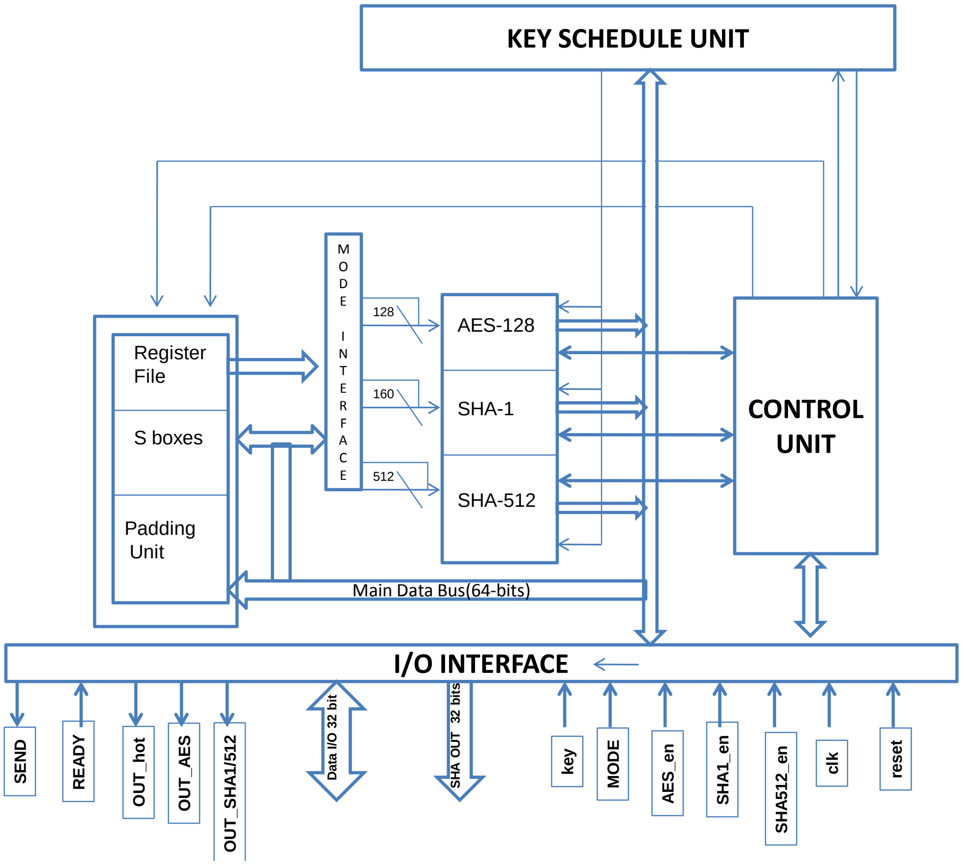


Figure 50. Main architecture of HSSec cryptographic coprocessor (From [62]). The control unit manages data processing and communication with the outside world. Cryptographic primitives (AES-128, SHA-1, and SHA-512) are arranged in a parallel orientation and use a common 64-bit global data bus. The key-scheduler block is used for key expansion and generating message schedules. The memory block consists of a register file, padding unit, and S boxes. The mode interface is responsible for modifying the input to the cryptographic primitives. The key scheduler performs the RotWord and SubWord transformations described above. The key scheduler also provides constants needed by the hash functions: SHA-1 uses a sequence of eighty, constant 32-bit words, and SHA-512 uses a sequence of eighty, constant 64-bit words.

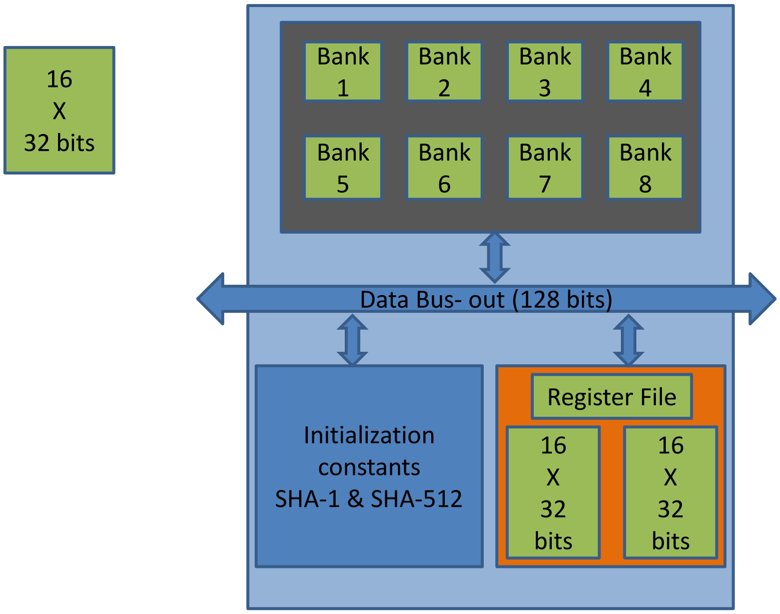


Figure 51. Organization of the memory block (From [62]). The memory block consists of three main parts. The first part is a set of registers used for storage of the initialization values required by the three cryptographic algorithms. The second part is a general-purpose register file used for storing temporary values that can be accessed quickly. The third part is the padding unit.

| Signal | Description |
|---|---|
| READY | Manages the synchronization of the coprocessor in order to receive data. It also controls the flow of data. |
| SEND | 1. Manages the synchronization of the coprocessor in order to send data. It also controls the flow of data.<br>2. Used also as a halt signal. |
| AES_en | Selection of AES-128 cryptographic algorithm. |
| SHA1_en | Selection of SHA_1 cryptographic algorithm. |
| SHA512_en | Selection of SHA_512 cryptographic algorithm. |
| MODE | Selection of mode CBC or ECB for AES-128. |
| OUT_hot | Determines which cryptographic algorithm is being used to coordinate the control of the output. |
| OUT_AES | Determines which cryptographic algorithm is being used to coordinate the control of the output. |
| OUT_1/512 | Determines which cryptographic algorithm is being used to coordinate the control of the output. |
| KEY | Key indication |
| Clk | Clock signal |
| Reset | Reset signal |

Table 6. Control signals of the HSSec cryptographic coprocessor (After [62]).

### 5. Use Scenario

The growing use of web services has resulted in an increased demand for confidentiality, integrity, and availability of data. One way to address these requirements is the use of the cryptographic coprocessor. In the following use scenario, we will explain how crypto coprocessors help provide security. More specifically, we will explore how credit-card transaction security is enhanced with the use of a cryptographic coprocessor. Billions of transactions occur daily across the world over the web. A client sends critical personal data such as credit-card information and transaction amount when purchasing an item. An attacker able to compromise the server can access the sensitive data and use it for a variety of nefarious purposes:

- The attacker can use the credit-card information to conduct future transactions.
- The attacker can replay the same transaction multiple times.
- The attacker can increase the amount of the transaction.

All these threats motivate a customer to conduct business on well-established sites only. Suppose that a customer wants to purchase a bike that is cheaper on a less established site.

With the use of a crypto coprocessor, the hypothetical transaction described above can be more secure. In this scenario, the cryptographic coprocessor resides on the server. The cryptographic coprocessor protects the personal information and transaction data by transmitting the data through a secure cryptographic channel to the customer's system. All sensitive personal-transaction data are encrypted when they are outside the server's domain. Neither an adversary nor an operator of the server can change the transaction data, and the transaction can execute securely [72].

### 6. Cryptographic Algorithm Performance

According to Schneier and Whiting, the principal criterion when designing or selecting a cryptographic application should always be security [73]. However, in the real world, high performance of the application is desirable. Therefore, the designer must balance the tradeoffs of security, performance, and usability of a cryptographic application [75]. Therefore, increasing a system's security may require compromising its usability and/or performance.

Below we present performance figures of AES-128, SHA-1, and SHA-512 in terms of speed, throughput, and power consumption from the literature. Our objective is not to determine which cryptographic algorithm is most efficient; rather, we will use the performance figures for making design decisions for our proposed 3DIC data-transformation system. Moreover, according to Scheiner et al., "it is very difficult to compare cipher designs for efficiency, and even more difficult to design ciphers that are efficient across all platforms and all users" [73].

#### *a. SHA-1 and SHA-512 Performance*

We first present performance data on SHA-1 and SHA-512 from speed benchmarks from the Crypto C++ library [74]. These speed benchmarks are based on algorithms implemented in C++, compiled with Microsoft Visual C++ 2005 SP1 (whole program optimization, optimized for speed), and executed on an Intel Core 2 1.83 GHz processor running Windows Vista in 32-bit mode [74].

| Algorithm | MB/Second | Cycles Per Byte |
|---|---|---|
| SHA-1 | 153 | 11.4 |
| SHA-256 | 111 | 15.8 |
| SHA-512 | 99 | 17.7 |

Table 7. Encryption rate in MB per sec and cycles per byte for SHA-1, SHA-256, and SHA-512 (From [74]).

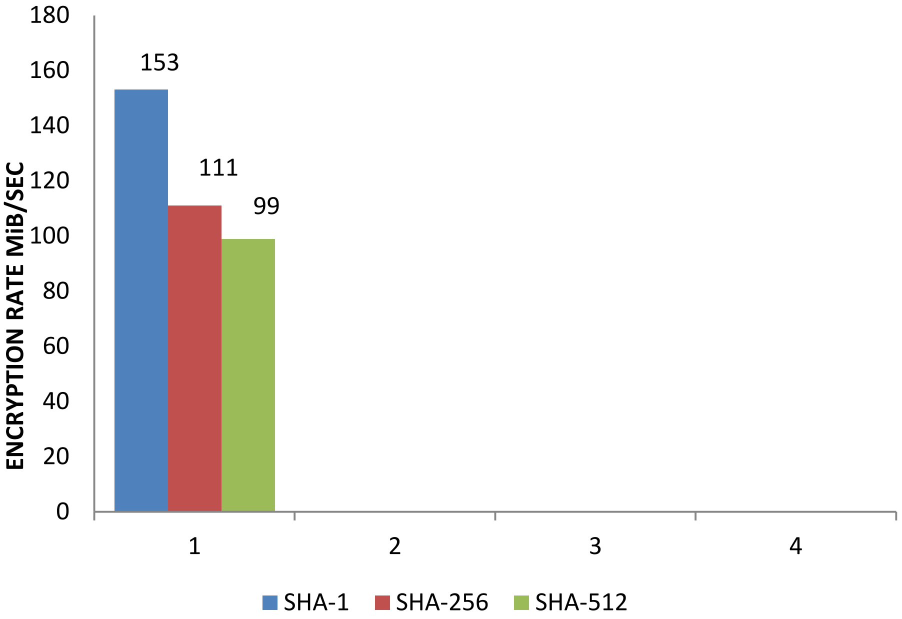


Figure 52. Encryption rate in MB per sec for SHA-1, SHA-256, and SHA-512 (From [74]). SHA-1 has the largest encryption rate, followed by SHA-256 and SHA-512.

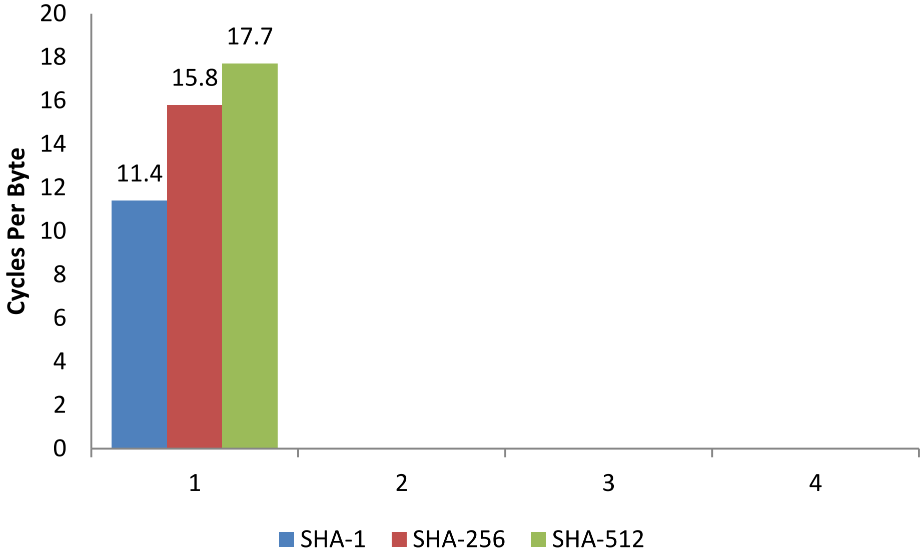


Figure 53. Number of cycles per byte for SHA-1, SHA-256, SHA-512 (From [74]). SHA-1 has the smallest number of cycles, followed by SHA-256 and SHA-512.

As explained above, SHA-1 generates a 160-bit message digest, and SHA-512 generates a 512-bit digest. SHA-512's longer hash value increases its resistance against a brute-force attack, compared to SHA-1's shorter hash value. However, SHA-1 is faster than SHA-512.

Next research compares the performance of different encryption algorithms, including SHA-1 and SHA-512, implemented within the .NET framework [75]. The .NET is an integral part of various applications running on Windows platforms and provides common functionality for those applications. It consists of a library and supports various programming languages The .NET framework's base-class library provides various elements such as user interface, data access, database connectivity, cryptography, web application development, and network communications [76].

Prior research measured the effect on the performance impact on MD5, SHA-1, and SHA-512 of varying the data size from 4KB to 135KB and 1MB [75]. Specifically, Dhawan measured performance in terms of requests per second for different user loads (different numbers of users) for different data sizes [75].

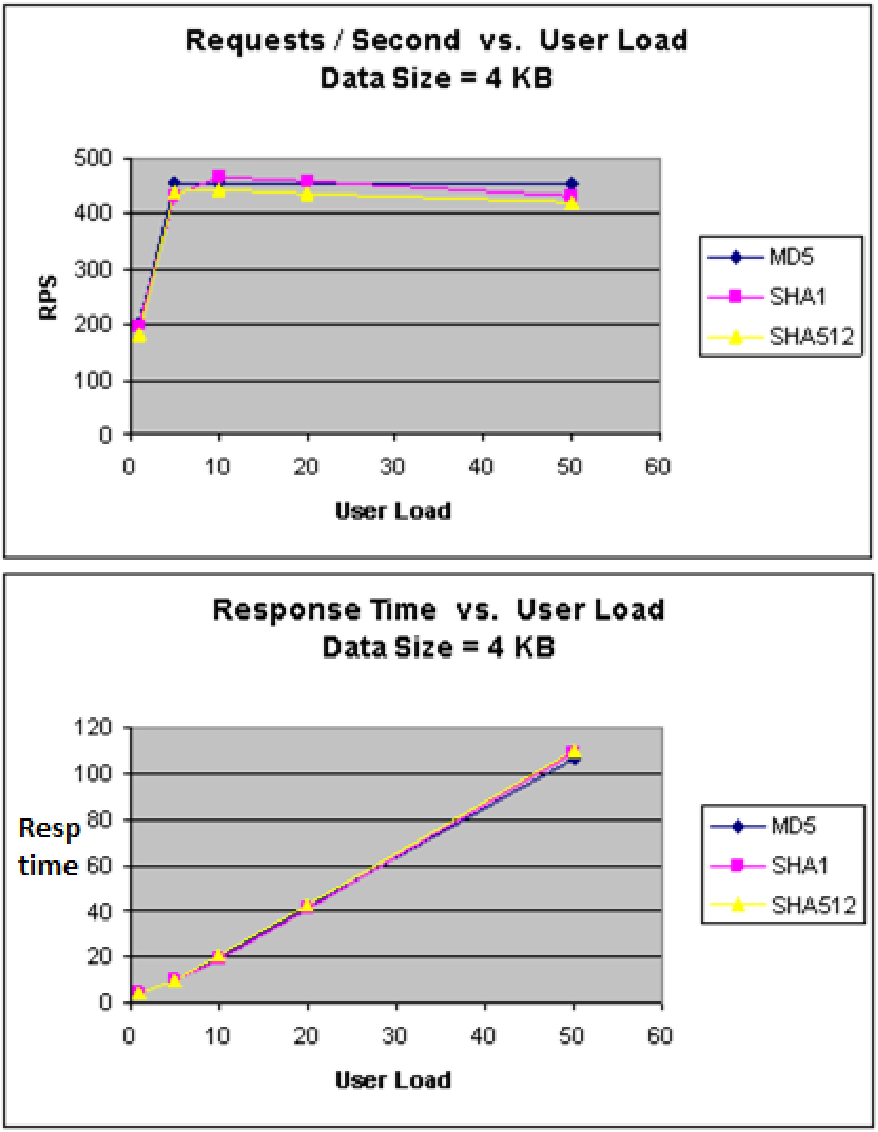


Figure 54. Hash algorithms (From [75]). MD5, SHA-1, SHA-512 for a data size of 4 KB: requests per second (RPS) and response time. All three algorithms have nearly the same performance, with SHA-512 being slightly slower.

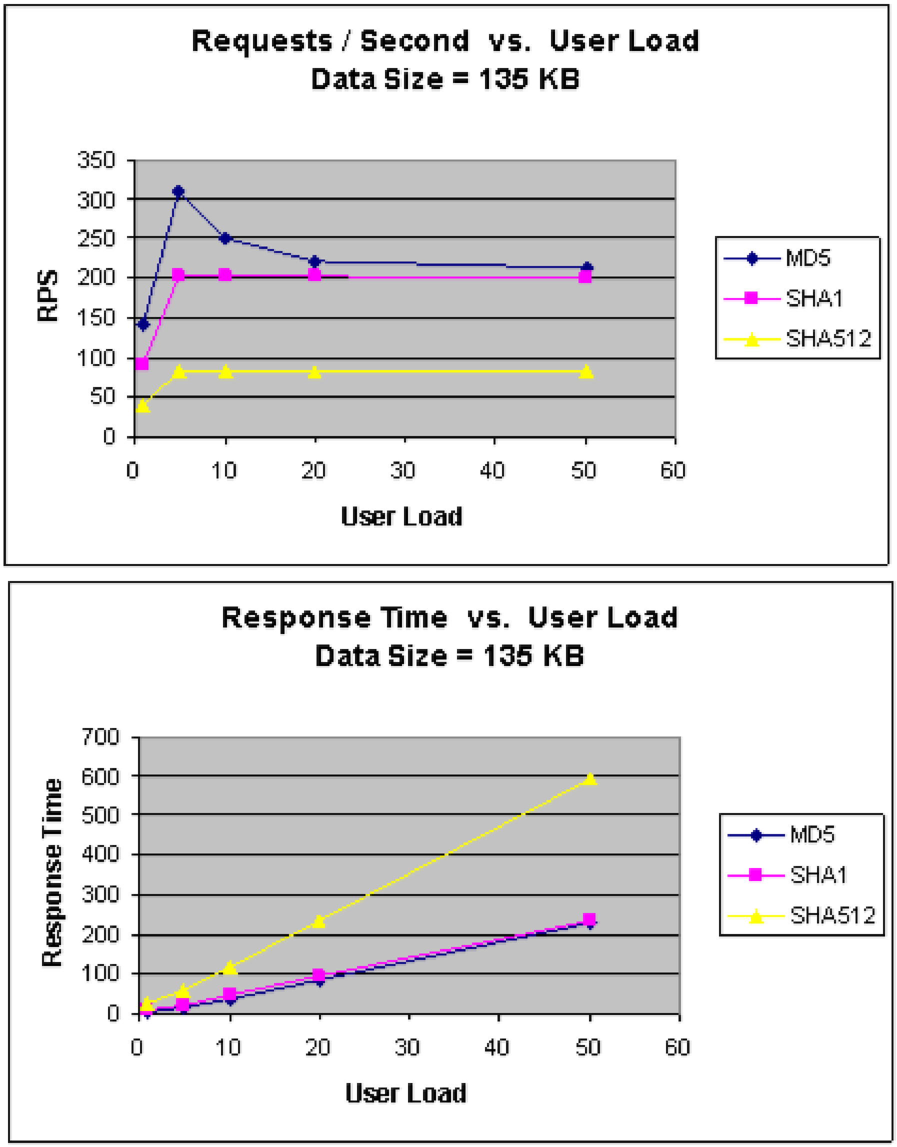


Figure 55. Hash algorithms (From [75]). MD5, SHA-1, SHA-512 for a data size of 135 KB: Requests per second (RPS) and response time. As the data size increases to 135 KB, there are more variations in the speed. For five users, as SHA-512 is almost 55% slower than SHA-1, and SHA-1 is almost 33% slower than MD5.

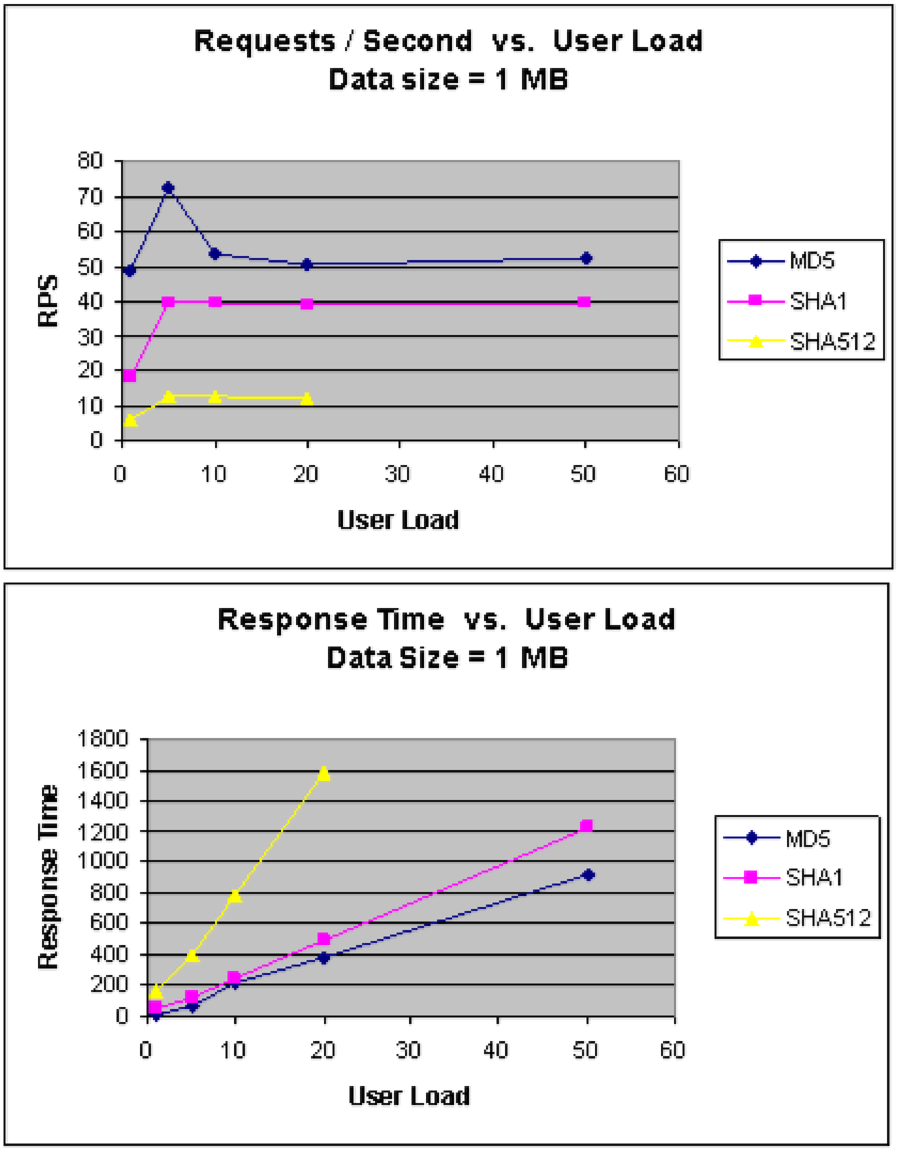


Figure 56. Hash algorithms (From [75]). MD5, SHA-1, and SHA-512 for a data size of 1MB: requests per second (RPS) and response time. As the data size increases to 1 MB, there are more variations in speed. For five users, SHA-1 is almost 72% faster than SHA-512.

These experiments show that larger message digests reduce the performance of the hash algorithms. This is an example of a tradeoff between security and performance.

Gladman explores the performance of a family of hash algorithms in terms of cycles per byte on Intel and AMD systems for different data lengths, from 1 to 100,000 bytes [77].

| Data Length | 1 | 10 | 100 | 1,000 | 10,000 | 100,000 |
|---|---|---|---|---|---|---|
| AMD64 (64 bit mode) | | | | | | |
| SHA1 | 672 | 70.1 | 13.07 | 9.79 | 9.4 | 9.7 |
| SHA224 | 1436 | 145.3 | 27.9 | 21.1 | 20.4 | 20.4 |
| SHA256 | 1483 | 149.9 | 28.4 | 21.1 | 20.4 | 20.4 |
| SHA384 | 1864 | 187.9 | 19.9 | 13.9 | 13.5 | 13.4 |
| SHA512 | 1939 | 195.6 | 20.6 | 14.0 | 13.5 | 13.4 |

Table 8. Cycles per byte for the family of SHA algorithms on an AMD 64 system (From [77]).

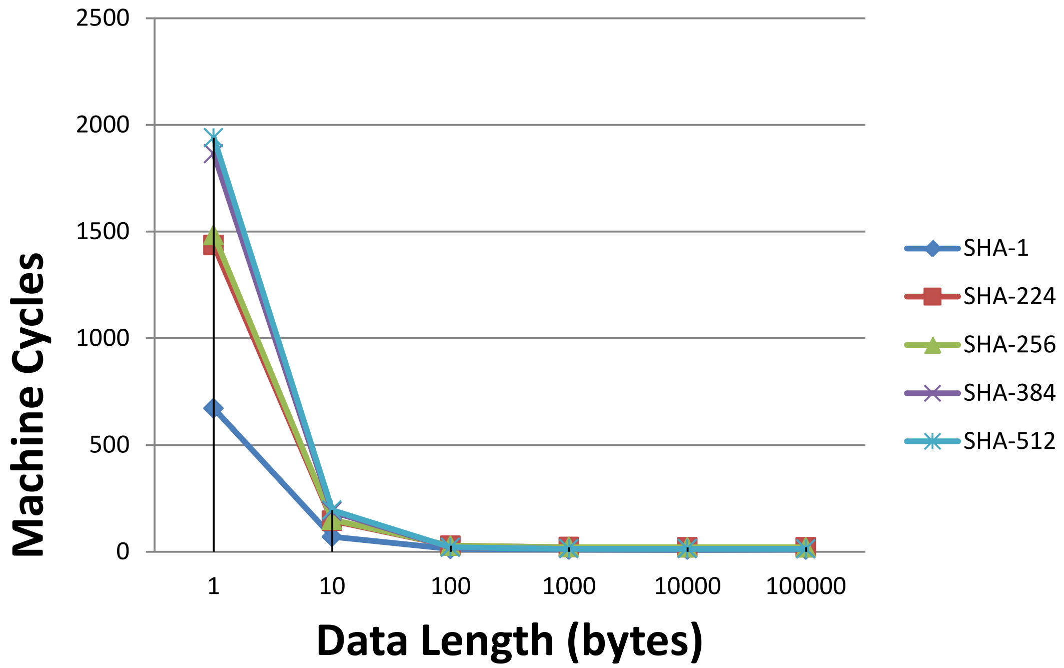


Figure 57. Performance of the SHA family of algorithms on an AMD 64 system (From [77]). For SHA-1, hashing one byte requires nearly 672 machine cycles, while SHA-512 requires 1939 cycles. Thus, SHA-1 is faster than SHA-512. Increasing the data length decreases the difference in speed, but SHA-1 is faster than SHA-512 in all cases.

| Data Length | 1 | 10 | 100 | 1,000 | 10,000 | 100,000 |
|---|---|---|---|---|---|---|
| Intel P3 | | | | | | |
| SHA1 | 1401 | 128.1 | 22.9 | 20.5 | 20.2 | 17.2 |
| SHA224 | 2865 | 294.1 | 59.4 | 42.7 | 41.4 | 41.0 |
| SHA256 | 2993 | 292.5 | 55.8 | 42.7 | 41.5 | 41.0 |
| SHA384 | 23253 | 2380.1 | 241.9 | 177.9 | 174.5 | 173.1 |
| SHA512 | 23653 | 2433.7 | 239.2 | 177.5 | 174.7 | 172.8 |

Table 9. Cycles per byte for the SHA family of algorithms on an Intel P3 system (From [77]).

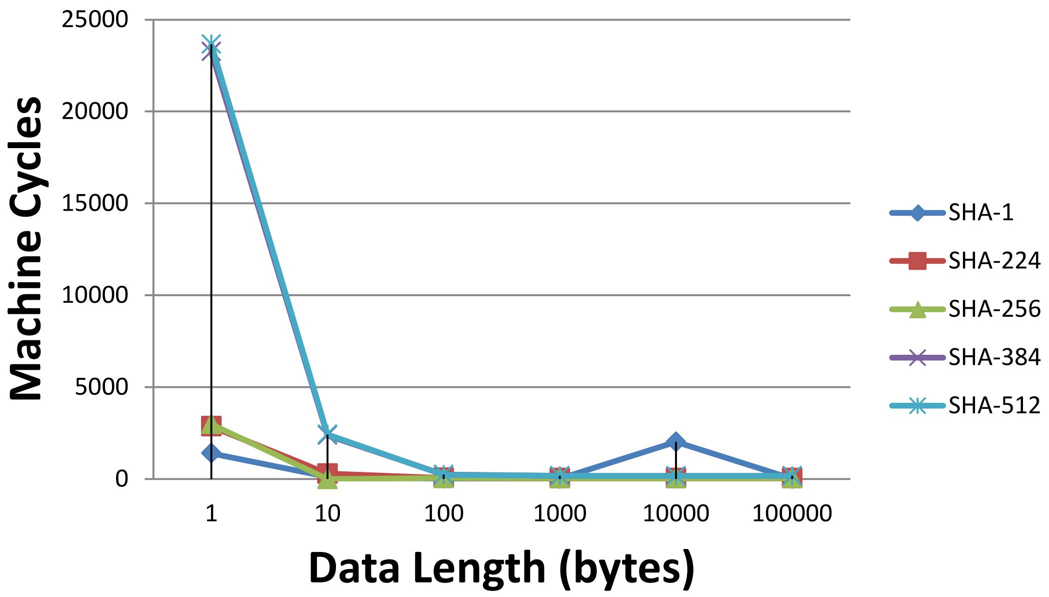


Figure 58. Speed of the SHA family of algorithms on an Intel P3 system (From [77]). SHA-1 requires nearly 1401 machine cycles to hash one byte, while SHA-512 requires 23653 cycles. Thus, SHA-1 is faster than SHA-512. Increasing the data length increases the difference in speed, but SHA-1 is faster compared to SHA-512 in all cases.

These experiments demonstrate that SHA-1 is faster than SHA-512. However, SHA-512 is much more resistant to brute-force attack, because it generates a 512-bit hash value instead of a 128-bit hash value. This is an example of a tradeoff between security and performance.

#### *b. AES-128 Performance*

For AES-128, we present performance figures from previous research in terms of speed, time, and energy consumption.

We present speed data for AES (128,192,256) with electronic codebook (ECB), cipher-block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR) modes from Crypto++ [74]. These figures reflect the performance of AES implemented in C++, compiled with Microsoft Visual C++ 2005 SP1 and executed on a 1.83 GHz Intel Core 2 processor running Windows Vista in 32-bit mode [74]. Figures 59 and 60 show that AES-128 is faster than AES-192 and AES-256. Also, AES/CTR mode (128-bit key) is the fastest configuration, with a encryption rate of 139 MB/sec. The slowest configuration is AES/CBC (256-bit key) with an encryption rate of 80 MB/sec.

| Algorithm | MB/Second | Cycles Per Byte |
|---|---|---|
| **AES/CTR (128-bit key)** | 139 | 12.6 |
| **AES/CTR (192-bit key)** | 113 | 15.4 |
| **AES/CTR (256-bit key)** | 96 | 18.2 |
| **AES/CBC (128-bit key)** | 109 | 16.0 |
| **AES/CBC (192-bit key)** | 92 | 18.9 |
| **AES/CBC (256-bit key)** | 80 | 21.7 |
| **AES/OFB (128-bit key)** | 103 | 16.9 |
| **AES/CFB (128-bit key)** | 108 | 16.1 |
| **AES/ECB (128-bit key)** | 109 | 16.0 |

Table 10. Encryption rate in MB per sec and cycles per byte for AES-128, AES-192, and AES-256 in conjunction with electronic codebook (ECB), cipher-block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR) modes (From [74]).

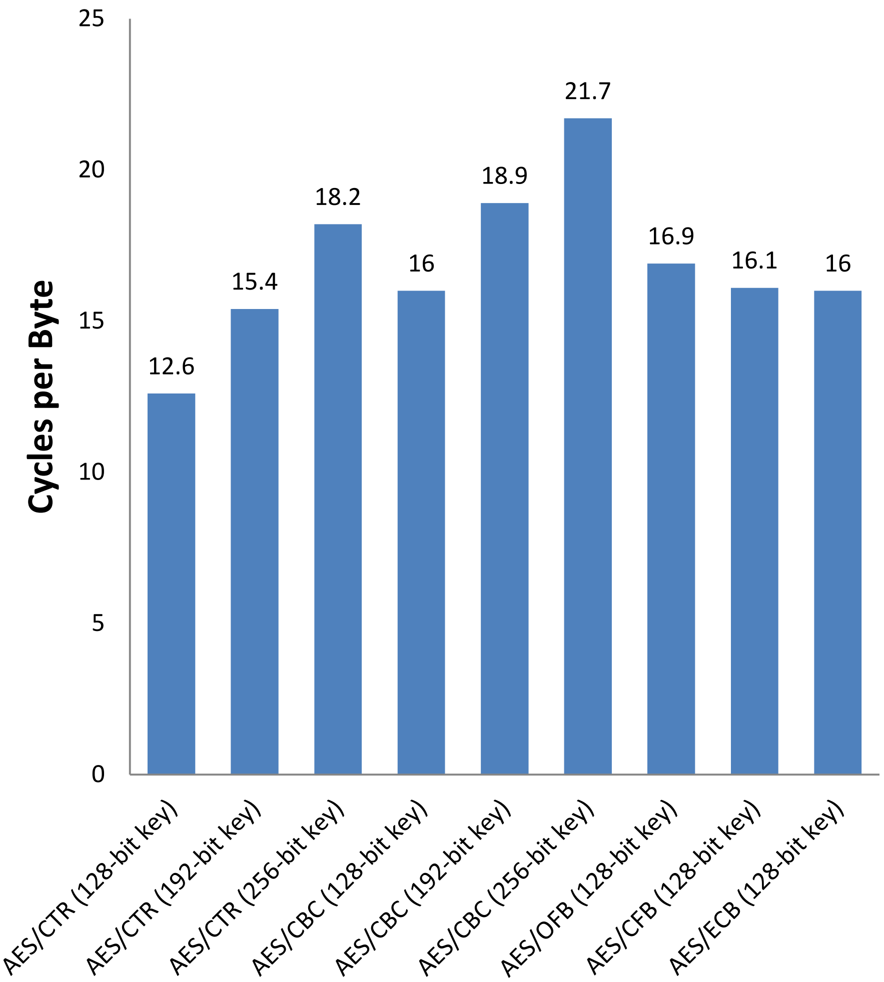


Figure 59. Number of cycles per byte for various combinations of AES key lengths and block cipher modes (From [74]). AES-128 for every mode is faster than AES-192 and AES-256. AES/CTR mode (128-bit key) is the fastest combination. All other combinations have performances ranging from 15.4 to 18.9 cycles per byte. The slowest combination is AES/CBC (256-bit key).

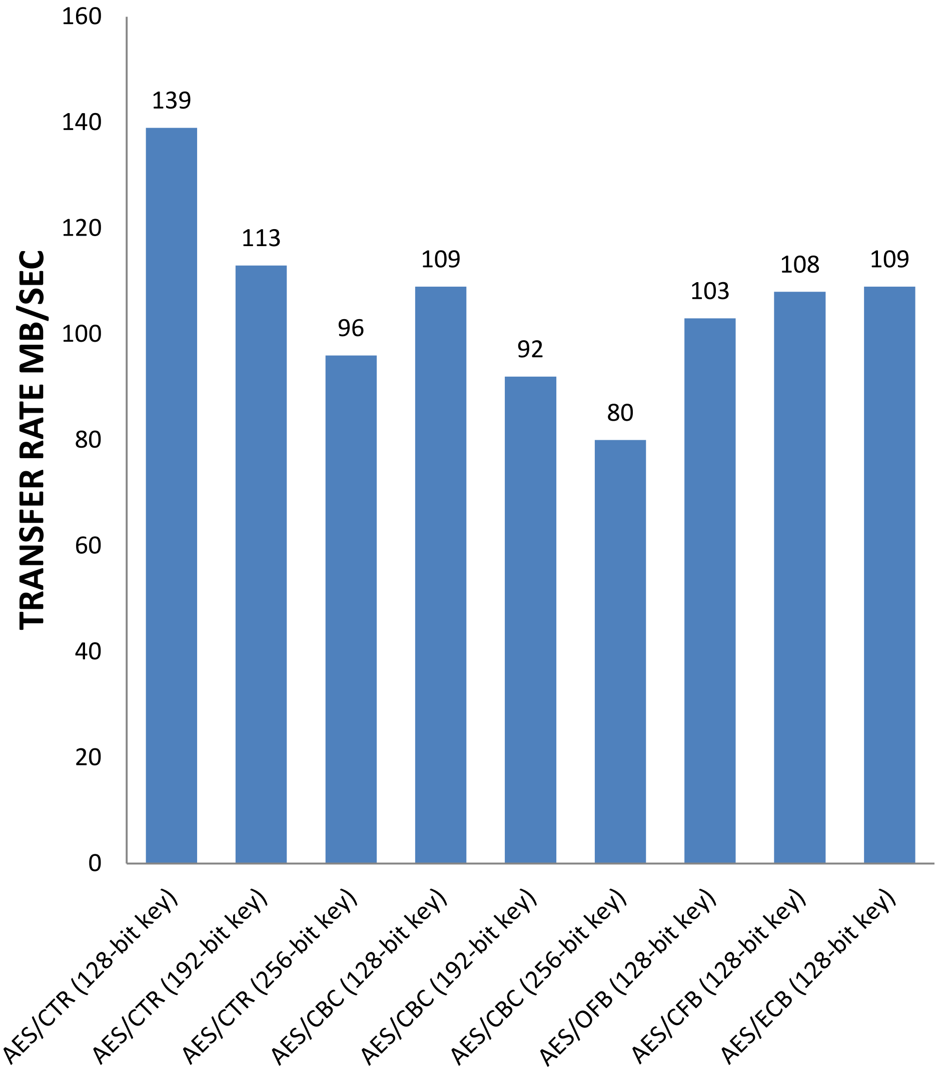


Figure 60. Encryption rate in MB/sec for various combinations of AES key lengths and block cipher modes (From [74]). AES-128 for every mode is faster than AES-192 and AES-256. AES/CTR mode (128-bit key) is the fastest combination, with an encryption rate of 139 MB/sec. The slowest combination is AES/CBC (256 – bit key).

Al Tamimi presents experimental results based on a C++ library implementing some of the most commonly used cryptographic algorithms [78]. AES-128, AES-192, and AES-256 were implemented in C++, compiled with Microsoft Visual C++ .NET 2003, and executed on a 2.1 GHz Pentium 4 processor running Windows XP SP 1.

This experiment explored the speed of AES for a data length of 256 MB. Figures 61, 62, and 63 show that AES-128 is faster than AES-192 and AES-256. Therefore, AES-256 sacrifices some performance for greater security.

| **Algorithm** | **Megabytes Processed** | **Time Taken(sec)** | **MB/sec** |
|---|---|---|---|
| **AES-128** | 256 | 4,196 | 61,010 |
| **AES-192** | 256 | 4,817 | 53,145 |
| **AES-256** | 256 | 5,308 | 48,229 |

Table 11. Performance of AES-128, AES-192, and AES-256 to process 256 MB of data in terms of CPU time and the encryption rate in MB/sec (From [78]).

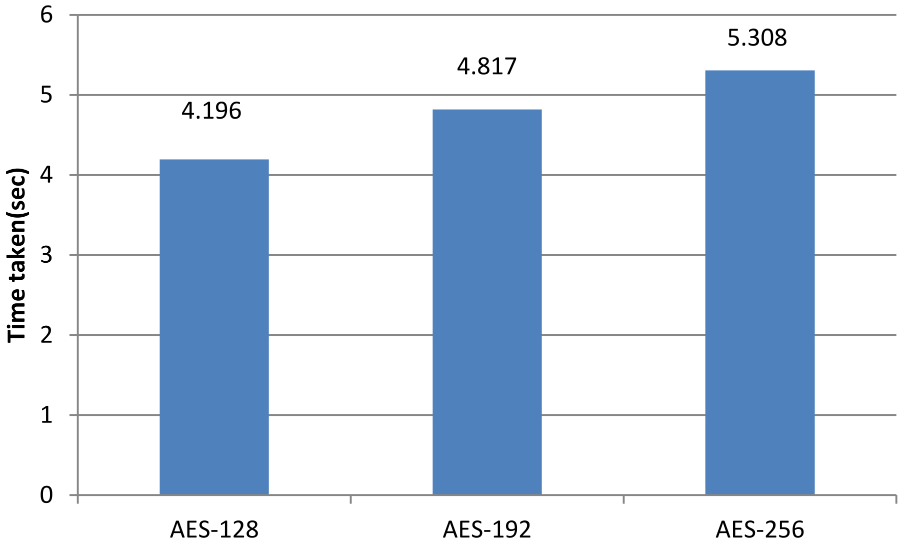


Figure 61. Time required for AES-128, AES-192, and AES-256 to encrypt 256Mb of data (From [78]). AES-128 requires the least time (4,196 sec) and AES-256 requires the most time (5,308).

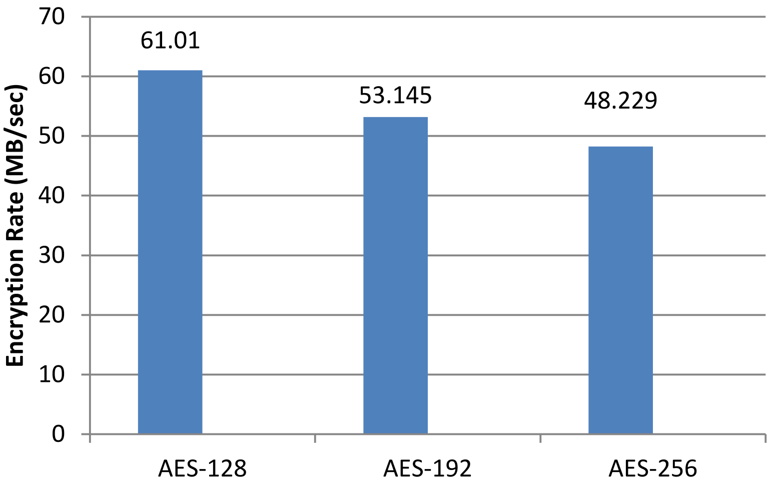


Figure 62. Encryption rate for AES-128, AES-192, and AES-256 to encrypt 256MB of data (From [78]). AES-128 has the largest encryption rate (61.01 MB/sec), and AES-256 has the smallest (48.229 MB/sec).

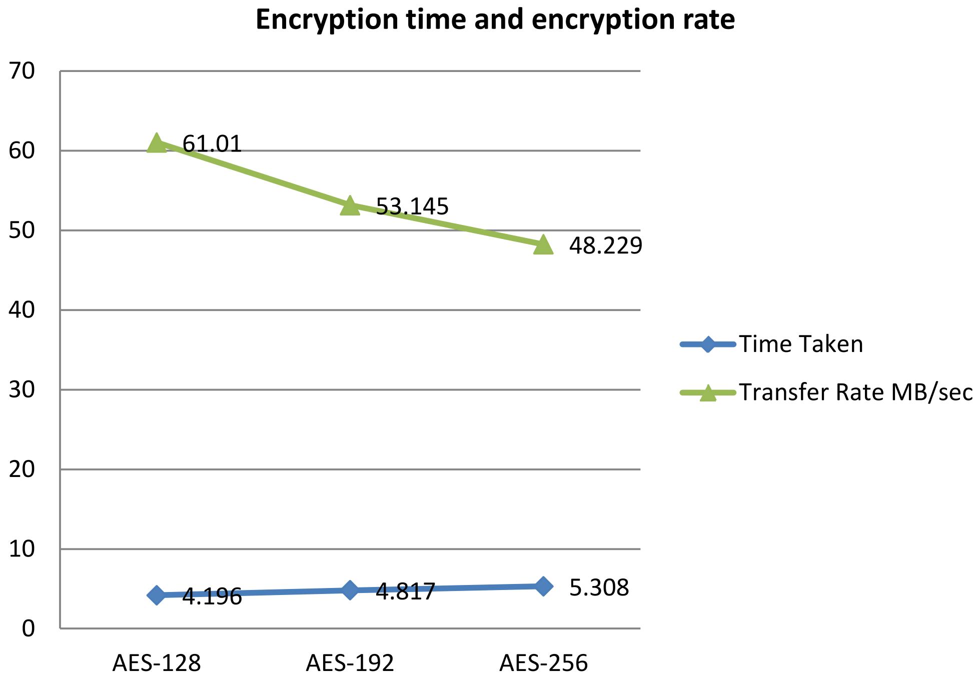


Figure 63. CPU time and encryption rate for AES-128, AES-192, and AES-256 to encrypt 256MB of data (From [78]). AES-128 has the highest encryption rate and the shortest time and is faster than AES-192 and AES-256.

In addition to speed, power is another important design constraint. For example, battery life is important for embedded systems supporting cryptographic applications. Cryptographic algorithms can be intense computations requiring substantial power consumption. Below we present two studies on power consumption for cryptographic applications, which show that the use of longer keys increases power consumption.

Hirani's dissertation explores the power consumption of various cryptographic algorithms, both symmetric and asymmetric, used by applications in wireless networks [79]. Hirani presents experimental results for AES for different key lengths (128, 192, and 256 bits). He first compares the changes in power consumption for different AES key lengths. Figure 64 shows that as the key length increases, so does power consumption. For AES-192, there is an increase of 8% in power consumption compared to AES-128, while AES-256 experiences an increase of 16% compared to AES-128. Therefore, increasing key length achieves greater security and resistance to attacks at the expense of an increase in power consumption.

| **Algorithm** | **Battery consumption (%)** |
|---|---|
| **AES-128** | 0.0041 |
| **AES-192** | 0.0044 |
| **AES-256** | 0.0047 |

Table 12. Battery consumption for AES for different key lengths (From [79]).

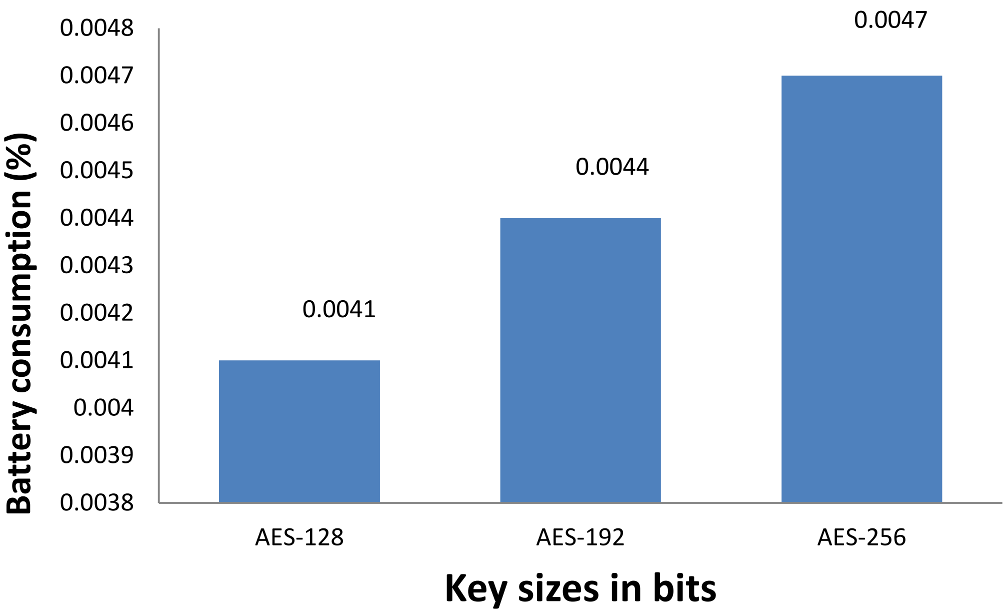


Figure 64. Battery consumption (%) for different AES key lengths (From [79]). AES-128 has the smallest power consumption, while AES-256 consumes the most power.

Potlapally et al. analyze the energy consumption of security protocols [80]. Their analysis is based on performing secure data transactions within a battery-powered system (Compaq iPAQ PDA). They perform measurements of the current drawn from the battery and measure the power consumed during each cycle of the cryptographic algorithm. They provide power consumption figures for AES-128, AES-192, and AES-256 in conjunction with electronic codebook (ECB), cipher-block chaining (CBC), cipher feedback (CFB), and output feedback (OFB). Their analysis also includes the key setup phase. Their analysis shows that AES-128 has the lowest power consumption for the key setup phase and for all block cipher mode configurations of AES. Also, CFB mode has the highest energy consumption of any AES operating mode, while ECB is the most economical in terms of energy consumption.

| Key Size | Key setup(μJ) | ECB (μJ/B) | CBC (μJ/B) | CFB (μJ/B) | OFB (μJ/B) |
|---|---|---|---|---|---|
| **128** | 7,83 | 1,21 | 1,62 | 1,91 | 1,62 |
| **192** | 7,87 | 1,42 | 2,08 | 2,3 | 1,83 |
| **256** | 9,92 | 1,64 | 2,29 | 2,31 | 2,05 |

Table 13. Energy consumption of various combinations of AES block cipher modes and key sizes, From [80].

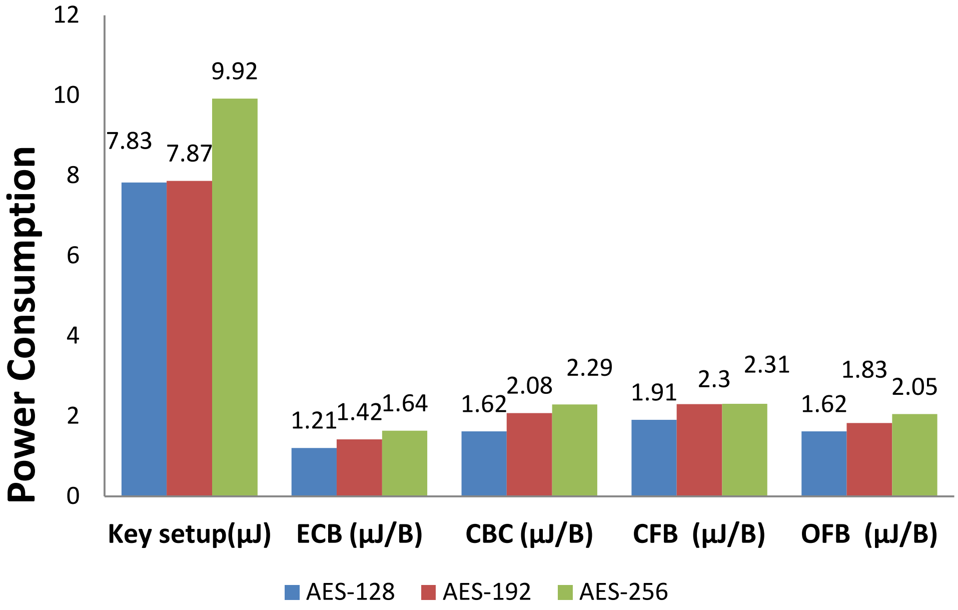


Figure 65. Power consumption of AES-128, AES-192, and AES-256 in key setup phase and in conjunction with electronic codebook (ECB), cipher-block chaining (CBC), cipher feedback (CFB), and output feedback (OFB) modes (From [80]). AES-128 has the lowest power consumption for the key setup phase and for all block cipher modes. Also, CFB mode has the highest energy consumption of any AES operation mode, while ECB has the lowest.

In conclusion, the designer of a cryptographic application must balance many performance constraints. For example, increasing the key length increases security, but also increases power consumption and processing time, while decreasing the encryption rate.

# V. 3D INTEGRATED CIRCUIT ARCHITECTURE

## A. INTRODUCTION

3D integration is an emerging integrated-circuit fabrication technology in which two or more IC dies are vertically stacked and connected with conductive posts. This allows a commodity die, or computation plane, to be combined with a custom die, or control plane. We envision a two-die system consisting of a general-purpose CPU in the computation plane and a data transformation coprocessor in the control plane.

In one application of 3D integration, a profile of signals in the computation plane is delivered at very high bandwidth to the control plane, where they can be compressed using an efficient hardware compression circuit. Once the data is compressed, it can then be sent off-chip for analysis over a lower bandwidth channel to a storage device. A major advantage of a 3D approach is reduced delay and increased bandwidth between the computation and compression functions, compared to a 2D implementation. This advantage can be applied to dynamic program analysis for reverse engineering of malicious software and post-mortem analysis of a system that has suffered an attack. The amount of data collected depends on the granularity of the signals collection and the speed of the system: collecting more signals results in a larger data stream. The high bandwidth between layers possible with 3D integration has the potential to increase the bandwidth to off-chip storage and to reduce the on-chip storage need.

Crypto processors are widely used in a variety of critical systems that require higher bandwidth encryption than that available with software encryption. They were initially developed for military cipher machines but have spread to smart cards, banking, telecommunications, networking, aerospace, and high-assurance computing platforms. A crypto coprocessor is a custom circuit for carrying out cryptographic transformations, often embedded in a tamper-resistant packaging. Systems will often combine a cryptographic coprocessor with a general-purpose processor, key storage, and other elements. A crypto coprocessor may implement just one algorithm or may support a variety of ciphers, e.g., DES, RSA, SHA-1, etc. To achieve the highest possible performance requires careful balancing of tradeoffs between speed, cost, power, and security during design and implementation.

## B. FACTORS IN 3D ARCHITECTURE

In this chapter, we present the main aspects that have to be considered when designing a 3DIC that requires different thinking, methods, and tools than a 2DIC. These factors include the bonding methods, the floor plan, power and ground networks, memory placement, thermal issues, and testing methods. We conclude with a “straw man” design for our own computation plane.

### 1. Bonding: Interconnection Methods

As discussed in Chapter II, in a three dimensional 3DIC, we have multiple layers which are stacked together. Various interconnect technologies [6], [81] can be applied to the 3D integration such as wire bonding, microbumps, through-silicon vias (TSVs), die-to-die, or contactless interconnection.

A common approach is wire bonding [81], where wires connect each die in a stack within a processor package. In this process, wires emerge from the I/O contacts on periphery of each die but are contained within the package. This approach is limited by the resolution of wire bonders and becomes more complicated as the number of I/O contacts in the chips increase. Wire bonding is not considered a true 3D technology because it does not provide the spatial locality advantages of TSVs, due to the fact that signals passing between layers must travel to and from the peripheries of their respective dies.

Another interesting approach is microbump technology [81]. This process uses either solder or gold bumps, placed on the surface of the die, to provide the required connections. The pitch of these bumps varies from 50 to 500 μm, although in some cases smaller sizes are possible. The mechanical stresses applied during the assembly process are significantly lower than wire bonding. Since the bumps only require the top one or two metal layers, the remaining layers are available for routing and devices. This specific 3D-packaging approach provides an enhanced vertical interconnect density compared to the wire-bonding approach. On the other hand, it does not reduce in a significant way the parasitic capacitance, because as, is stated by Davis et al., “a microbump bonded cube must still route signals to the periphery before sending them back to the destination inside the cube” [81]. This 3D packaging method enables one or more chips, fabricated under diverse technologies, to be joined into a single unified stack.

According to Davis et al., through-silicon via interconnection offers the greatest interconnect density, but this comes at a higher cost [81]. TSVs (through silicon vias) enable the highest vertical interconnect density. TSVs are short vertical wires used to connect planar wires. In face-to-face bonding, one layer is placed face down onto the second wafer, which is facing up. Alternatively, layers can be stacked face to back, but this requires TSVs to pass through the bulk silicon. Note that face-to-face bonding only allows a maximum of two layers in the stack; face-to-back bonding is required for more than two layers. Face-to-face vias are smaller than the TSVs used in the face-to-back bonding process; however, TSVs are still required to support I/O through the top layer. The maximum number of layers that can be stacked in a face-to-back configuration is an open research question. Needless to say, more layers in the stack present greater challenges and higher costs. Note also that dies can be bonded in a wafer-to-wafer, die-to-die, and die-to-wafer fashion. Furthermore, TSVs can be manufactured using a via-first approach, where the vias are made prior to making the devices and the wires; or via-last, where the vias are added after the devices and wires are made. Via-first is more expensive than via-last but allows smaller via sizes.

The process of making via-last involves drilling holes from the upper wafer to the lower and then filling the holes with tungsten to provide connectivity. As described by Loh et al., current fabrication technologies are capable of providing die-to-die via pitches within a range of 10 μm x 10 μm to 1 μm x 1 μm [6]. Figure 66 depicts a cross-sectional view of the die-to-die interface. The die-to-die vias [17] are placed on the top of the metal stack of each die and are bonded after alignment. They are differentiated from I/O pads, and their size and electrical characteristics are similar to vias used to connect on-die metal routing layers. Last year, IBM managed to reduce the via pitch to 0.2 μm x 0.2 μm through silicon-on-insulator (SOI) technology [82]. With this technology, as described by

Davis et al., we can "avoid the need for passivating the hole by polishing the substrate away completely, down to the buried oxide" [81].

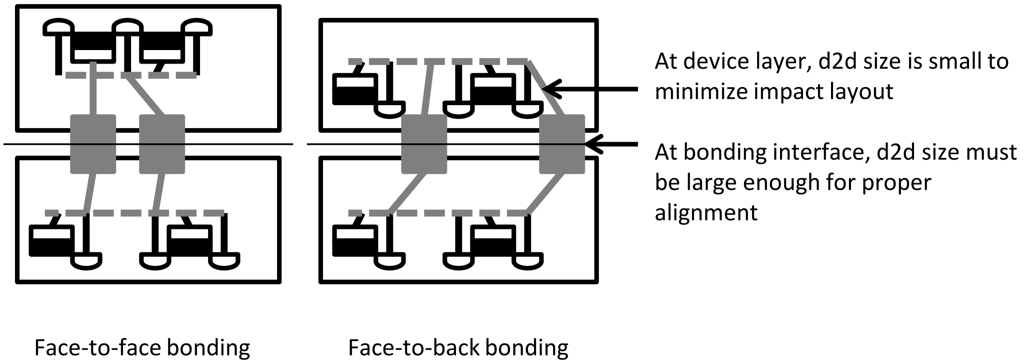

At device layer, d2d size is small to minimize impact layout
At bonding interface, d2d size must be large enough for proper alignment
Face-to-face bonding
Face-to-back bonding

Figure 66. Cross-sectional view of the die-to-die interface for face-to-face and face-to-back bonding arrangements (From [6]). According to the position of the metal layers of the upper die relative to those of the lower die, the bonding process is either face-to-face, where the metal layers of the two die face each other, or face-to-back, where the metal layers of the lower tie touch the bulk silicon of the upper die [6].

Another method of connecting 3DIC layers is contactless interconnection, which involves the use of capacitive or inductive coupling for the communication between layers [81]. As described by Davis et al., "this approach eliminates the processing steps for creating inter-layer DC connectivity and eliminates the need to route signals to the periphery, allowing for reduced wire lengths" [81]. Also, due to the fact that the contactless approach requires only a minimum amount of processing for chip thinning, which consequently minimizes the complexity of fabrication process, the manufacturing cost is significantly less as compared to the manufacturing cost of approaches that use microbumps and through-silicon vias.

Figure 67 illustrates the interconnect technologies described above.

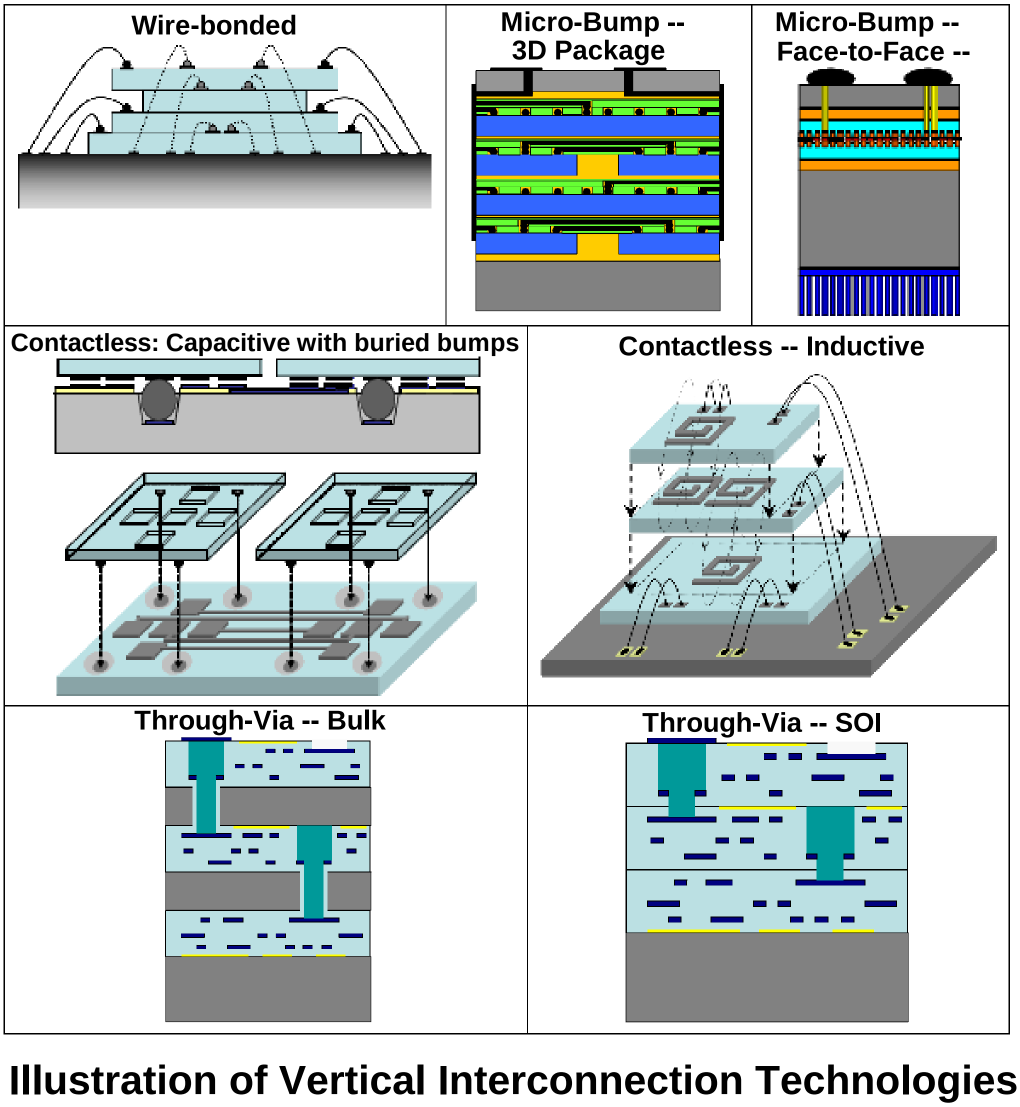

Wire-bonded
Micro-Bump -- 3D Package
Micro-Bump -- Face-to-Face --
Contactless: Capacitive with buried bumps
Contactless -- Inductive
Through-Via -- Bulk
Through-Via -- SOI
Illustration of Vertical Interconnection Technologies

Figure 67. Illustration of Vertical Interconnect Technologies (From [81]). Wire bonding (top left): wires connect each die in a stack. Micro bumps (top middle and top right): solder or gold bumps, placed on the surface of the die provide the required connections. Contactless (middle row): involves the use of capacitive (middle left) or inductive (middle right) coupling for communication between layers, Through-silicon via (TSVs) (bottom row): Short vertical wires between layers of interconnect, used to connect the planar wires. Their size varies from 50 µm to 1µm. With the implementation of silicon-on-insulator (SOI) technology (bottom right), the pitch of vias reduced to 0.2 µm x 0.2 µm.

### 2. Manufacturing Methods

There are two primary methods of manufacturing three-dimensional (3D) chips with respect to wafer level stacking: the “bottom-up” and the “top-down” fabrication methods [83], [85].

The bottom-up wafer-fabrication method [84], [85] builds a multi-die processor in a manner similar to how multistory buildings are constructed. Each die has several internal layers for devices, interconnects insulation etc. For 3DICs built using the bottom-up method, the first die is constructed and each of the layers is laid down. Next, the second die is constructed, along with its layer, and so on. According to Euronymous, a significant drawback of this method is that it is difficult to make changes to the design of one of the die without affecting the entire stack [84]. However, a significant advantage of this method is that the size of the inter-layer vias can be reduced because the size of the transistor devices is reduced.

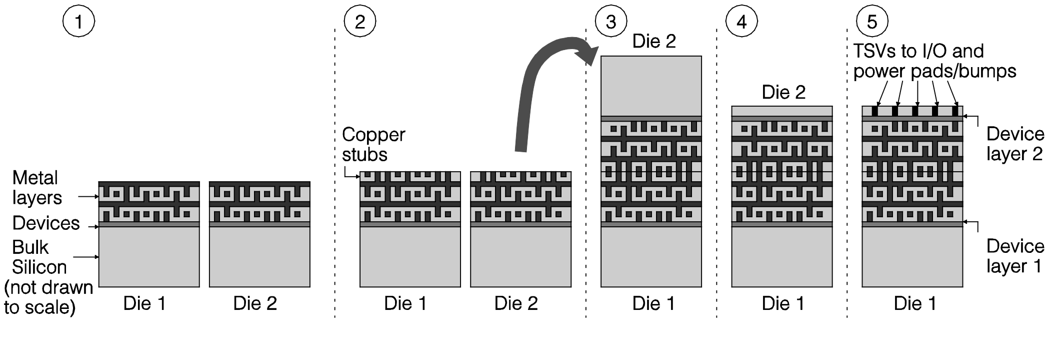
With the top-down wafer fabrication method [84], [85], each die is manufactured separately, and all the manufactured dies are bonded together at the final stage. This method has significant advantages. First, the certification and testing process is more accurate, due to the fact that each die can be tested independently. Thus, the final step bonds layers that have already been tested and certified. Another advantage is that this method allows diverse layers, manufactured using heterogeneous processes and optimized to a specific purpose, to be joined into a single unified stack. For example, one layer could be optimized for computation and another for sensing light. However, a disadvantage of the top-down method is that the size of the inter-layer vias cannot scale with the transistor devices. Nevertheless, this fabrication method is less costly, due to the fact that it is easier to make changes to an individual die without affecting the rest of the stack.

The top-down method encompasses the face-to-face and the face-to-back methods. In the face-to-face method [84], the metal layers of each die are stacked facing each other, and their interconnect layers are connected using die-to-die vias. On the other hand, in the face-to-back method [84], all layers in the stack have the same orientation. The distance between layers is larger, and the TSVs are longer and thicker than die-to-die vias used in face-to-back bonding, as they must pass through bulk silicon to reach the metal layers of the next die. Face-to-back topology provides better scalability and can be adopted for architectures requiring more than two layers. In our proposed architecture, we have selected the face-to-face bonding process because we require only two layers.

### *a. Face-to-Face Bonding*

As it described by Loh et al. and depicted in Figure 68, the fabrication steps for a face-to-face, top-down construction are as follows [6]:

1. We have the two processed wafers.
2. Copper via stubs are connected to the top level metal areas of the dies.
3. After the face-to-face alignment of the two wafers, they are joined using thermo compression. The total area between two dies will be completely populated by die-to-die vias. These vias serve the following needs: they provide a path for I/O signals, power, and ground, they are good conductors for dissipating heat in the 3DIC, and they support the mechanical connection of the two dies.
4. With the use of chemical–mechanical polishing (CMP), one layer of the stack is thinned from 10 to 20 μm.
5. The thinning process allows the through-silicon vias (TSVs), which provide the external I/O signal, power, and ground connections, to be relatively short.



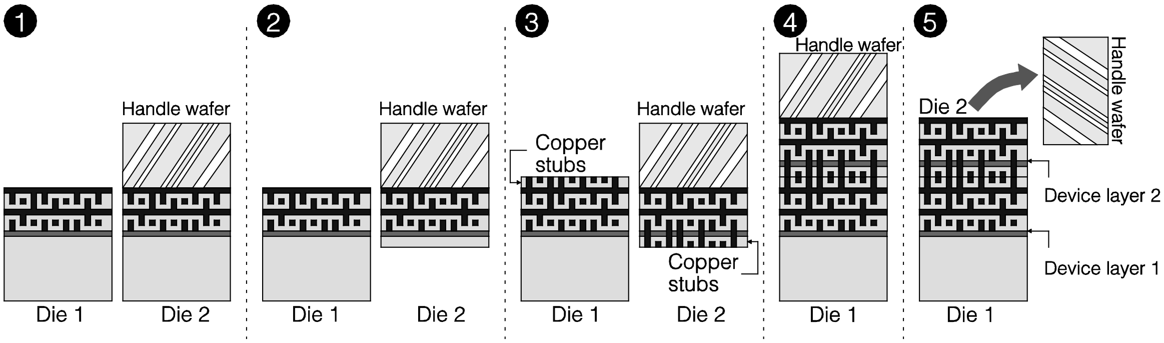
**Fabrication steps for face-to-face bonding**

Figure 68. Fabrication steps for face-to-face bonding (From [6]).

### *b. Face-to-Back Bonding*

As described in [6] and depicted in Figure 69, the fabrication steps for a face-to back, top-down construction are as follows:

1. We have the two processed wafers. Before thinning the wafer, it must be attached to a handle wafer.
2. With the use of chemical–mechanical polishing (CMP), the wafer is thinned to about 10 to 20 μm. The handle wafer provides mechanical support to the thinned wafer and prevents it from being broken.
3. The two halves of the die-to-die (d2d) vias are joined. The process of constructing the via stubs, with respect to the face wafer, is similar to the face-to-face process. With respect to the back wafer, the vias are etched in a way similar to the face-to-face vias, which provide signal, power, and ground.
4. The two dies are bonded together by thermo compression.
5. The thinned die is released from the handle wafer.



**Fabrication steps for face-to-back bonding**

Figure 69. Fabrication steps for face to back bonding (From [6]).

## C. FLOORPLAN, POWER, AND GROUND NETWORK

For 3DIC floor planning and power placement, several parameters have to be considered, the principal being thermal dissipation. Several tools for modeling placement that consider thermal effects have been developed, such as a tool developed by Cong et al. in 2004, which is a thermal-driven floor-planning algorithm for 3D ICs [85].

Gabriel H. Loh et al. developed a floor planner that “takes a micro-architectural net list and determines the placement of the functional modules while simultaneously optimizing for performance and thermal reliability. The traditional design objectives such as area and wire length are also considered” [86].

A proposal focusing on power/ground distribution and the IR drop effect (a voltage drop due to the resistance of the mesh) is presented by Falkenstern et al. [87] They use a B*-tree for floorplan representation; to represent the power and ground, they use a resistive mesh and a simulated annealing engine at the end. They form some interesting conclusions; for example, the average IR drops generally decrease if the designers increase the number of layers in the 3DIC, thus reducing the area in each layer. Because the modules are closer, taking advantage of the small horizontal distance provided by the stacking process, the power/ground edges are shorter, resulting in small IR drops. Also 3DIC allows a better distribution of modules, thus reducing the number of modules consuming energy from the same edge, allowing each edge to have less current, therefore reducing the IR drop. Those points are important because smaller IR drops increase the performance of the circuit [87].

Gabriel H Loh et al. present some conclusions based on simulations [6]. In their calculations, about 30% of the die-to-die vias are used for power and ground. For face-to-back topologies, normal pins can be used for supplying off-chip power, but when face-to-face topologies are used, the power supply from the board to the chip must be delivered using TSVs. This does not present serious concerns, “because the inductance of a single 10-mm-wide TSV is less than 2.5 pH for a single return path. Many return paths exist in a full chip, which further reduces the effective inductance. This additional inductance has little effect compared to the switching noise observed in the on-die power distribution networks of existing processors” [6].

Suppose we have a 3DIC with half the footprint of a 2DIC. The 3D chip will probably use half the pins to transport power, compared to the 2D chip, doubling the current on those pins. Even if the 3D chip had the same number of pins, the designer probably would have to double the current to support two layers of circuit. Loh et al. present conclusions based on simulation about the number of pins and power distribution that illustrate these concepts [6]. First, they conclude that the TSV can easily support the current increase and explain that a 3D IC does not consume the same amount of power as a 2D IC. Thus, having half the number of pins does not double the current density in each pin, because a 3D IC can reduce the power requirements due to shorter distances between on-chip modules [6].

## D. MEMORY

Memory implementation is a significant design issue and an important design decision in our 3DIC architecture. Many memory issues have been explored in the recent literature. Although we are not proposing to stack memory in our 3D architecture, we want to consider this issue for future analysis and improvements to our design that will provide the benefits of an on-chip cache. These issues include limits on the pins on both ends of the memory controller and the DRAM modules, as well as motherboard area requirements [92]. One option that was studied was to implement a method of direct vertical stacking of many dies of memory, one above another, all connected through TSVs [88], [89]. In this process, all memory dies would be constructed separately, utilizing either 2D SRAM or DRAM. This method offers simplicity, and only minor changes are required during the manufacturing process, because it is done in sequential steps. The gain in performance is due to the fact that on-chip buses are faster, consume less power, and are less capacitive [92]. IBM estimates that a 60% latency reduction is possible by using an on-chip DRAM [91]. Results of simulations, using Simple Scalar 4.0 in comparison with a baseline 2DIC with a 3GHz CPU, 750 MHz memory, 1MB L2 cache, and 8MB L3 cache, show an average speed up of 126% over 2D implementation for floating-point programs and a 59% speedup for integer programs.

Loh et al. [92] say that the above implementation does not receive a large performance benefit, due to the small size of the workload; therefore, they also propose implementing a 64-byte bus to memory, which, by itself, increases the performance to 71.8% over the 2DIC.

An observation may be made about DRAM in 3D architecture [93] during the refresh operation. Due to the greater operational temperature of a 3DIC, it is reasonable to estimate that the refresh rate needed to retain data will also increase, and a proposed solution is the use of smart refresh, which uses a counter to refresh the required memory rows and banks only, saving a geometric mean of 6.87% of total energy.

Another interesting approach is that of Sun et al., who explore the adoption of a coarse-grained 3D partitioning method focused on 3D DRAM design. The main purpose of the above method [90] is to share the global routing of the memory address and data bus between all DRAM dies. To achieve this, they used coarse-grained TSVs with a pitch in the tenths of μms. In this method, a partition of individual memory subarrays is created and, once they are split, distributed to all available dies. Each bank of memory is divided into sub-banks, where each sub-bank is divided into 3D subarray sets, where each one contains *n* 2D subarrays, including the required memory calls and the peripheral circuits. Using the above method in each 3D subarray set, the required 2D subarrays share only address and data I/O TSVs. Therefore, the total number of TSVs is reduced, and the global addresses can be distributed across the total number of dies, achieving the optimum result.

Loh et al. explore another similar approach focused on implementing a large L2 cache memory. They adopted a coarse-grained method to place a cache above one or more processor cores [6]. By placing cache memory on top, they reduced the number of TSVs or face-to-face vias. An enhancement uses a banked implementation, in which each bank can be the same as it is in a conventional 2DIC, but in the case of a 3DIC, each bank is stacked on top of another. The main advantage of this method is that the global routing can be reduced significantly. The two implementations described above are shown in Figure 70.

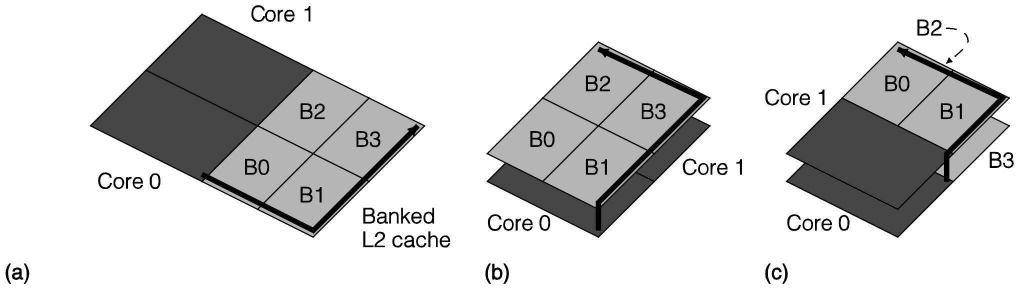


Figure 70. Implementing a cache in 3D (From [6]). (a) A baseline 2D processor with L2 cache; (b) an L2 cache stacked above the cores; (c) L2 cache banks stacked on each other. In Figure (1)(c), each bank can be the same as it is in a conventional 2DIC, but in the case of a 3DIC, the banks are stacked. The advantage is that the global routing can be reduced significantly. The bold black arrow in each subfigure illustrates the reduction in interconnection length.

Loh et al. propose other approaches geared towards a more aggressive 3D memory organization that are beyond the scope of this thesis [92].

## E. THERMAL

Many advantages of 3DICs arise from the reduction in overall wire length. Unfortunately, this does not come free.

According to Puttaswamy et al., temperatures on 3DICs are higher than conventional 2DICs for three reasons [94]. First, the 3DICs suffer from a higher power density because the active devices are stacked vertically. Second, heat dissipation is less effective in 3DICs, because the temperature gradients are lower. Also, the physical path used for dissipation becomes significantly longer along the vertical dimension towards the heat sink. Finally, the area of the die in contact with the heat spreader is not large enough because the effective area (footprint) of each die is minimized. This leads to less efficient dissipation to the heat sink. Various techniques have been developed to address thermal issues in 3DICs. By selecting an optimal topology, it is possible to achieve a thermal profile similar to that of a conventional 2DIC. For example, memory can be stacked above the processor core [17]. This significantly reduces the number of main memory accesses and corresponding bus activity, which leads to a reduction in power consumption, and a decrease in the thermal impact. Black et al. investigate three options for 3D memory stacking on a base processor die (Intel Core TM 2 Duo microprocessor), in which cores have private, level-one instruction and data caches of 32KB and share a 4MB level-two cache (L2). The first option is to increase the L2 size from 8MB to 12MB of static random-access memory (SRAM). This implementation places the additional 8MB L2 cache on top of the base processor die. The second option is to replace the L2 SRAM with a larger L2 dynamic, random-access memory (DRAM), thus replacing the 4MB L2 cache with a 32MB stacked L2 DRAM. The third option stacks a 64MB DRAM on top of the base processor. All the above options illustrated in Figure 71.

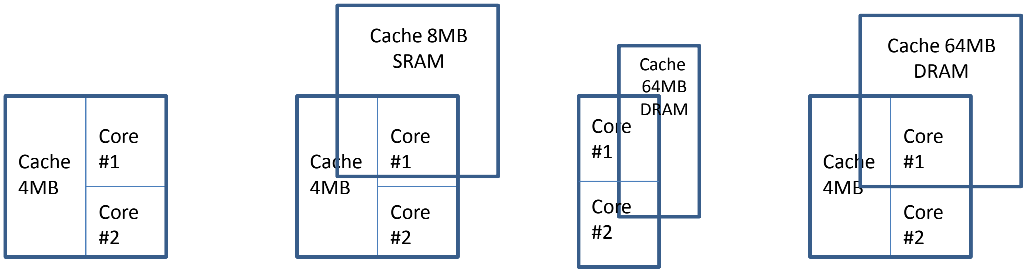


Figure 71. Memory-stacking options (After [17]): (a) 4MB baseline; (b) 8MB stacked, for a total of 12MB, with an increase of the L2 size from 8MB to 12MB of static random-access memory (SRAM); (c) 32MB of stacked DRAM with no SRAM, replacing the L2 SRAM with a larger L2 dynamic random-access memory (DRAM), thus replacing the 4MB L2 with a 32MB stacked L2 DRAM; and (d) stacking a 64MB DRAM on top of the base processor.

Thermal analysis of the proposed options demonstrates that the thermal impact due to stacking memory is insignificant compared with its performance and power advantages, as shown in Figure 72.

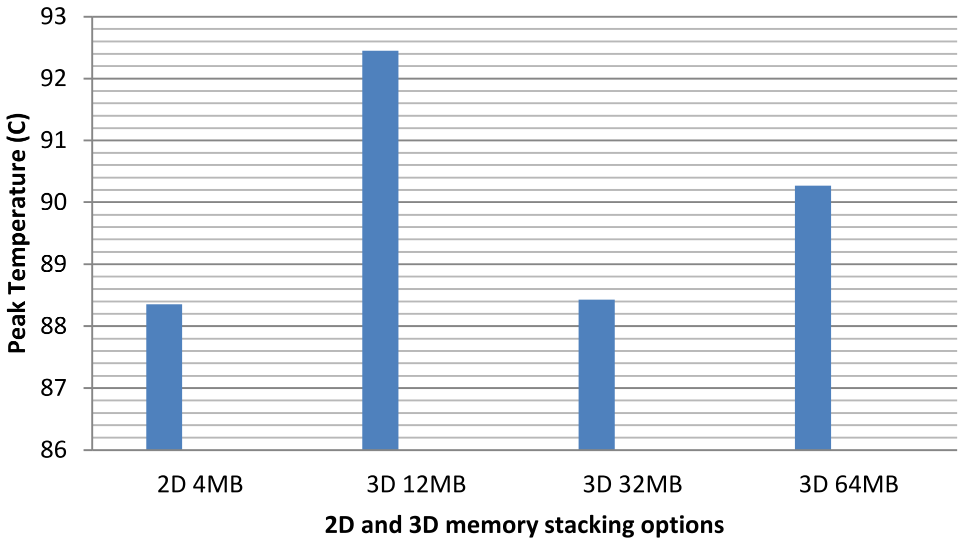


Figure 72. Temperature results for the stacked 12MB, 32MB, and 64MB memory options compared to the baseline 4MB (After [17]). The thermal impact of stacking memory is slightly greater than 2DICs.

Bryan Black at al. analyze and compare the power, frequency, thermal, and performance factors of a 2D architecture [17]. They try different combinations and conclude that:

- By limiting the temperature to that of a 2D architecture, a 3DIC can achieve an 8% increase in performance with 34% reduction in power consumption, due to distance and latency optimization.
- By limiting the performance to that of a 2D architecture, a 3DIC achieves a reduction in power consumption of 34%.
- By limiting the frequency to that of a 2D architecture, a 3DIC realizes an increase in the temperature of 14$^{o}$C and a performance increase of 15%.

Frequency and temperature play an important role in any 3D architecture. Most solutions for thermal issues utilize careful floor planning and a small reduction in processor speed, which does not reduce the overall advantage of a 3D architecture, as described by Loh et al.

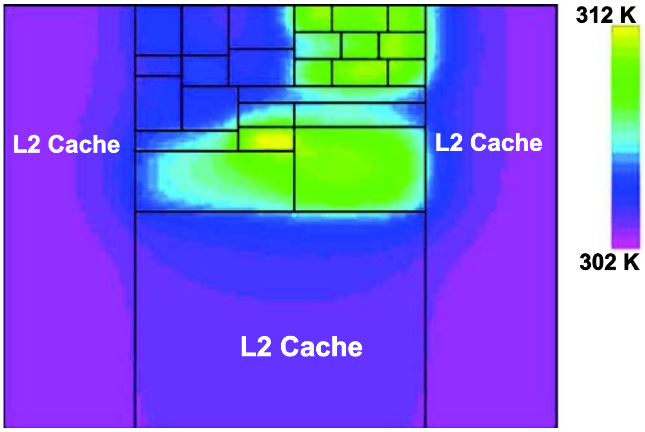
“It is found that, in spite of the lower operating frequency of a 3D chip (as imposed by thermal concerns), the overall system performance can still be significantly better than conventional planar designs, especially for memory intensive applications” [101].

The approach of Cong and Zhang is to increase the thermal conductivity of the stack by inserting thermal vias [95]. According to Sapatnekar, “the temperature may also be reduced by improving the effective thermal conductivity of paths from the devices to the heat sink. An effective method for achieving this is through the insertion of thermal vias: thermal vias are structurally similar to electrical vias, but serve no electrical purpose. Their primary function is to conduct heat through the 3D structure and convey it to the heat sink” [96]. However, according to Hua et al., implementations using thermal vias do not consider the fact that they can increase routing congestion, which consequently leads to the use of longer interconnects and thus to a significant increase in dynamic power [97]. The increase in dynamic power can result in higher temperatures and power leakage. In [97], Hua et al. explore the mapping between dynamic power and leakage power in two designs, by altering the number of layers and related number of thermal vias. They used two case studies involving low-power and high-performance applications and evaluated the tradeoff described above. Their research concluded that the overuse of thermal vias does not significantly affect the 3DIC system performance from the increase in wire length. In the case of low-power applications, the thermal effect is not significant. In the case of high-performance applications, adopting a specific process of placement of thermal vias, it is possible to significantly reduce the thermal effects.

Some other interesting approaches to dealing with thermal issues involve the rearranging of heat sources [96]. The locations of the heat sources can be moved through careful placement of components. Floor planning is one of these techniques. Hang et al. explore a floor-planning algorithm that reduces the peak temperatures of a 3DIC [99]. This algorithm is divided into two stages. First, it determines an optimum partitioning of the functional blocks into layers, decreasing the total wire length. Next, it reconfigures the floor plan of the layers that did not fully compact during the first stage.

Another interesting research is the floor-planning algorithm of Li et al, which determines the optimum floor plan and placement of thermal vias. Their process is performed in two steps. First, all blocks are distributed to layers, and then the number of vertical thermal vias required for each layer is determined. Second, the floor-planning process is performed to determine the optimum floor plan for each layer and the number of horizontal thermal vias required on each layer. Their method achieves a reduction in thermal vias of 15% and increases the usable area and wire length.

Adopting all the countermeasures against thermal effects described above, we can maintain the advantage of 3DICs. Puttaswamy and Loh studied the thermal behavior of a high-performance microprocessor built with two die and four die in a 3D technology and showed that the temperature increases are not as much as was previously thought. Techniques such as via layers, copper metallization, and modern packaging materials increase efficiency. Finally, 3D implementations of one conventional processor have thermal profiles similar to the 2D implementation. Figures 73, 74, 75 illustrate these findings.



Thermal Profile of the Planar Processor

Figure 73. Thermal Profile of the Planar Processor (From [94]).

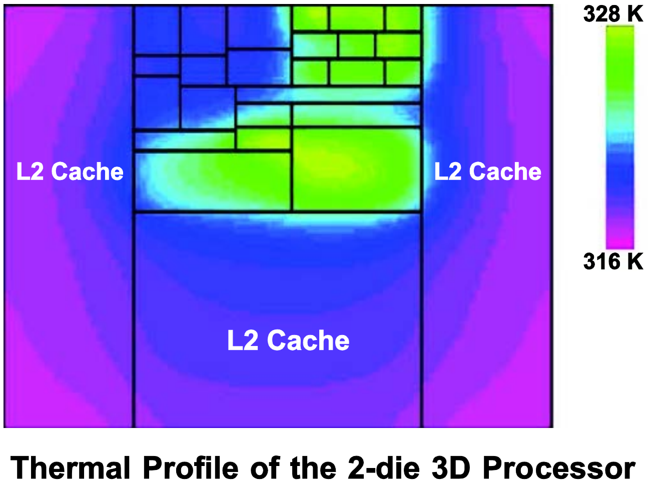


Figure 74. Thermal Profile of the Two-Die, 3D Processor (From [94]).

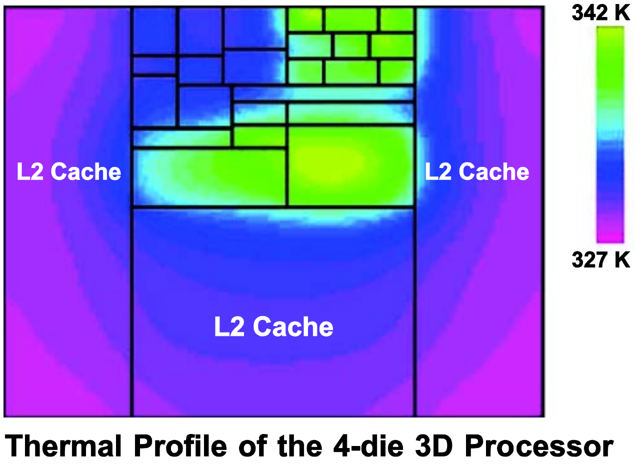
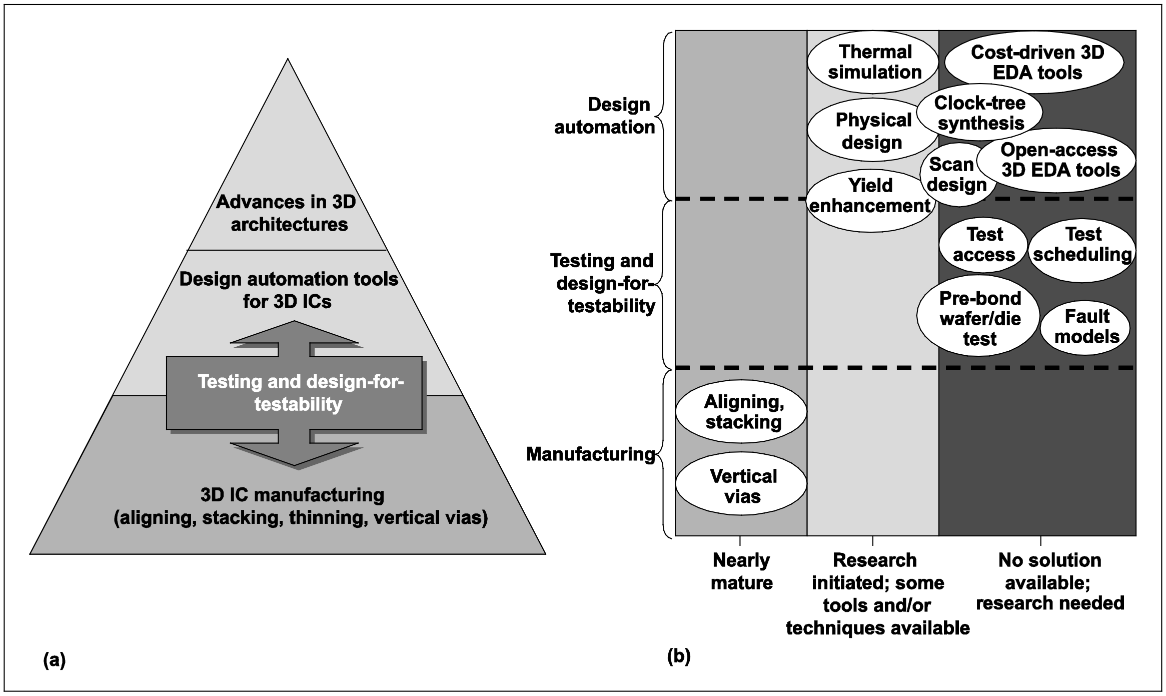


Figure 75. Thermal Profile of the four-die, 3D Processor (From [94]).

## F. TEST

According to Xie, "One of the potential obstacles to 3D-technology adoption is the insufficient understanding of 3D testing issues and the lack of DFT solutions"[15]. This is due to the fact that in 3D technology, the test probing needles from the probing cards cannot access inside the wafers. Other challenges include thermal issues, alignment, bonding, and thinning. Figure 76 illustrates testing challenges in 3DIC design and the status of several 3DIC research challenges.


Advances in 3D architectures
Design automation tools for 3D ICs
Testing and design-for-testability
3D IC manufacturing (aligning, stacking, thinning, vertical vias)
(a)
Design automation
Testing and design-for-testability
Manufacturing
Thermal simulation
Cost-driven 3D EDA tools
Physical design
Clock-tree synthesis
Scan design
Open-access 3D EDA tools
Yield enhancement
Test access
Test scheduling
Pre-bond wafer/die test
Fault models
Aligning, stacking
Vertical vias
Nearly mature
Research initiated; some tools and/or techniques available
No solution available; research needed
(b)

**3D IC testing and related challenges in the context of 3D integration: Role of 3D IC test (a); status of 3D IC R&D (b).**

Figure 76. A) The role of 3DIC testing in the development process. B) status of 3DIC research (From [102]).

Lee and Chakrabarty describe problems in face-to-face bonding: "The bottom die has up to hundreds of thousands of copper pads, but their small size and large number make probing of signals difficult. The top wafer would be hard to be probed from the copper side, the TSVs are buried and C4 bump pads are not fabricated prior to bonding" [102]. With face-to-back bonding, "the top die is more testable than the bottom because the C4 bump pads can be fabricated on the top layer. However, the top wafer must be thinned, which introduces the problems of ultrathin wafer processing and limits the ease with which the wafer can be probed. Typically, the probe card applies weight in the range from 3 to 10 g per probe. Therefore, the probe weight per wafer can be as high as 60 to 120 kg, which is a serious issue for thinned wafers" [102]. To solve these issues, research is underway using techniques described by Lee and Chakrabarty, such as "contactless testing and proximity I/O based on near-field wireless communication, inductive coupling, and capacitative coupling" [102].

Another significant factor in a 3D architecture is the cost and time to manufacture. Testing directly impacts in the total cost and time, as described by Grochowski et al.: “Integrated circuit (IC) testing for quality assurance is approaching 50% of the manufacturing costs for some complex mixed-signal IC’s” [103]. The techniques developed for 2D testing to reduce time and cost cannot be made efficient for 3D manufacturing, as described by Lee and Chakrabarty: “Modular testing, which is based on test access mechanisms (TAMs) and IEEE Std 1500 core test wrappers, provides a low-cost solution to the test access problem for a System on Chip (SoC); many I/O and scan terminals for the embedded cores can be accessed from a few chip pins. For today’s 2D ICs, several optimization techniques have been reported in the literature for test infrastructure design to minimize test time. Similar techniques are needed for 3D ICs, but we are now confronted with an even more difficult test access problem: the embedded cores in a 3D IC might be on different layers, and even the same embedded core could have blocks placed on different layers. Only a limited number of TSVs can be reserved for use by the TAM. Although many TSVs can be integrated in a 3D IC, most are required for power, clock, and signal lines, and the need for a ‘’keep out’ area requires optimization techniques that make judicious use of TSVs for test access. Wrapper design and optimization must also go beyond IEEE 1500 and consider how a core on multiple layers can be wrapped under TSV constraints” [102].

To address the testing problem, we adopt the solutions presented by Wu et al., using scan chains in two implementations to test the 3D-IC [104]. As scan chains can present challenges due to the time spent on testing, the length of the wires, and the area, most implementations divide the chip into smaller areas to be tested, reducing the testing costs.

The first technique applies a genetic algorithm (GA) to determine a best-path chain to map all the test points based on constraints, such as number of TSV, wire length, and scan time. The use of a GA, together with simulated annealing (SA), is the best tool to optimize multi-objective goals, with the advantage that a GA takes into account a pool of solutions to avoid local minima, as compared to just one solution computed by SA. This implementation uses three approaches, each with its advantages and disadvantages.

Approach one, Figure 77: each layer is considered an independent 2DIC, and using a 2D scan-chain tool, the chains are designed layer by layer, with a TSV at the end of one layer's chain linking to the beginning of the next layer's chain.

Advantage: use of a simple 2D chain tool to design the scan chains. The number of TSV is minimal (*n*-1 TSV for *n* layers).

Disadvantage: this method is optimal for each layer, but combining can lead to a suboptimal global design.

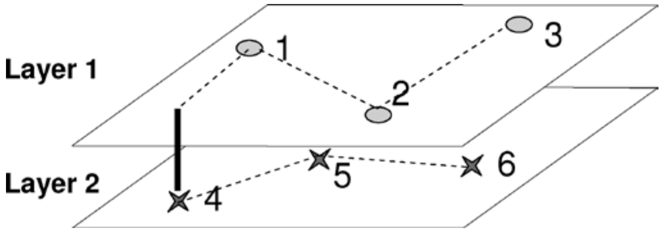


Figure 77. Approach One (From [104]). Two independent scan chains tied together by only one TSV.

Approach two, Figure 78: The cells to be scanned in all layers are projected onto just one plane, and a simple 2D chain tool is used to design the global scan chain. This approach does not take into account whether the cells are in different planes and will require TSVs, which can be a good design choice if the distance between layers is negligible.

Advantage: use of a simple 2D chain tool to design the scan chains.

Disadvantage: Since the TSVs are not considered, it can result in too many TSVs.

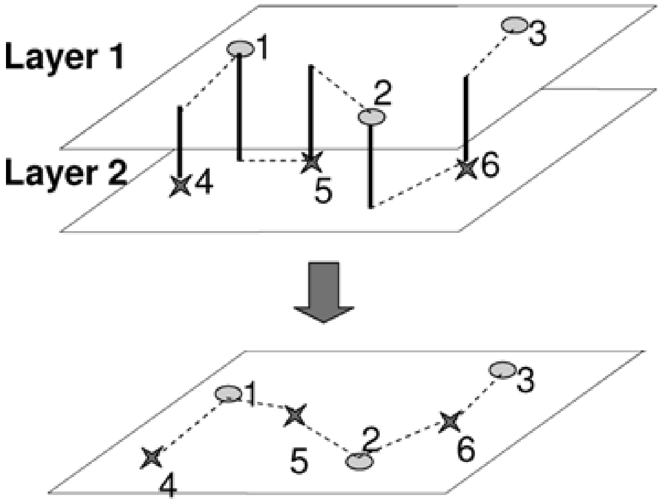


Figure 78. Approach Two (From [104]). All testing points are projected onto just one layer, and a 2D chain tool computes the scan-chain path.

Approach three, Figure 79: This approach computes the small global chain considering vertical and horizontal distances. The algorithm considers the horizontal Manhattan distance and the vertical distances.

Advantage: This is a true 3D scan chain that accounts for a globally optimal chain.

Disadvantage: The 2D chain tool has to be modified in order to address the 3D chain.

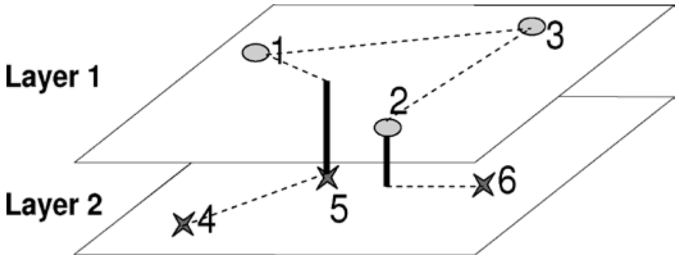


Figure 79. Approach Three (From [104]). This is a true 3D approach in which the tool computes the optimal path, considering horizontal and vertical distances.

The importance of these three approaches is that one of them is used as a prioritizing method (constraint) in the genetic algorithm for computing the scan chain, which determines the best scan chain design for testing.

GAs consist of five stages: (1) formation of the original population, a random list of “chromosomes” (characteristics of this population); (2) execution of several rounds until a condition is achieved; the rounds use a fitness function (a function applied over the chromosomes in order to calculate some specific requirement); (3) reproduction, which selects the new population based on some criteria over the result of the fitness function; (4) a crossover step in order to exchange characteristics of different chromosomes; and (5) mutation, which generates the new population for the next round.

The GA presented by Wu et al. uses integers from 1 to *N* to represent all the flip-flop cells to be tested, and the solution is a list of cells in the order visited, such that each node is visited just once [104]. This list is called the chromosome of the GA. The fitness function is calculated in order to determine the lowest wire length of the scan chain, so that after the reproduction phase, the best chromosomes are the ones with the lowest scan-chain wire length. In the crossover process, sections of each winning chromosome are exchanged to generate new chromosomes; if during this insertion a node appears twice, it is deleted from the original chromosome, the mutation takes place, and a second round begins.

The second proposed test technique uses integer linear programming (ILP), which is a minimization of an objective linear function under a linear constraint—in this case, minimizing the wire length of a scan chain, given the number of TSV as a constraint.

We will not present all the mathematical implementation of this technique, which is described by Wu et al. [104]; however, we will introduce the basic concept of the model evaluated using Xpress-MP, a commercial ILP solver.

In order to use this tool we first must specify the model and insert its parameters. The basic model of this scan chain ILP is shown below; the step that follows this specification is the definition of this model in Xpress-MP tool language. We assume that this model has a unique path starting on node *u*, traveling *N* nodes passing through nodes *i,j,* and ending in the *v* node; it also is constrained in the number of TSV, which has to be lower than *L*, as shown in Figure 80.

ILP Description, Figure 80:

1. For every cell *I,* there is only one immediate successor per scan cell, so the path $x_{ij}$ is equal to 1;

2. For every cell *j*, there is only one immediate predecessor per scan cell, so the path $x_{ij}$ is equal to 1;

3. There is no immediate predecessor for the initial scan pin *u*;

4. There is no immediate successor for the final scan pin *v*;

5. A scan cell cannot connect to itself.

6. *L* is a constraint on the number of TSV; each step between layers is considered equal to one (L2 - L1 = l = 1).

7. If the function is nonlinear, it is replaced by a new binary variable to ensure proper linearization.

8. Cell *i* is either before cell *j* or after cell *j* in the chain.

9–10. The initial node *u* is before every other node, and the end node *v* is after every other node in the scan-chain.

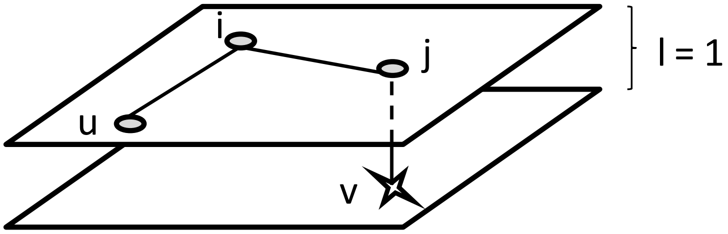


Figure 80. A visual representation of the ILP specification inserted to Xpress-MP.

The results from a comparison between the GA and ILP methods [104] demonstrated a small reduction in wire length using ILP, so we recommend that approach. Until no better tool is developed, the ILP method can be used for scan chain design with “near-optimal solutions” [104].

Our proposed architecture uses face-to-face bonding, which has testing and accessibility advantages as described by Emma and Kursun [105]. Their method uses scan chains in each layer that are accessible in a boundary scan. Their method also employs an additional infrastructure that allows a total 3D test chain.

THIS PAGE INTENTIONALLY LEFT BLANK

# VI. THE IDEAL 3D SYSTEM

## A. INTRODUCTION

The main advantage of having a fully functional computational plane and a control plane is that the computational plane can be manufactured in an untrusted foundry, and security-critical functions can be implemented in a separate die that is fabricated in a trusted foundry. The two dies can then be joined in a trusted facility. This allows dual use of the commodity-computation plane, which provides economic benefits. However, it requires some small changes to the computational plane.

For example, the computational plane has to provide clock signals to the control plane for synchronization, and this requires the availability of die-to-die connections. The extra cost of modifying the computational plane is amortized across all custom, 3D designs using that plane [4].

Now that we have presented some architectural issues, we define our proposed architecture in more detail. Our approach requires some changes in the computational plane, such as:

- Vertical clock-signal delivery to the control plane and clock buffers to synchronization
- Vertical connections for data transfer, from computational-plane registers to control-plane compression buffers
- Vertical posts for control/query signals among computational and control planes

Our proposed architecture does not require a memory connection; the compression and crypto devices have their own memories.

We propose a two-layer IC, with a computational plane and a compression-encryption plane stacked in a face-to-face architecture (providing the smallest possible distance between the layers [106]), allowing information to flow from the computational plane to the compression-encryption plane (control plane) as fast as possible, and with the die-to-die communication achieved using micro bumps that provide an enhanced vertical-interconnection density and smaller distance, as compared to the wire-bonded method [106].

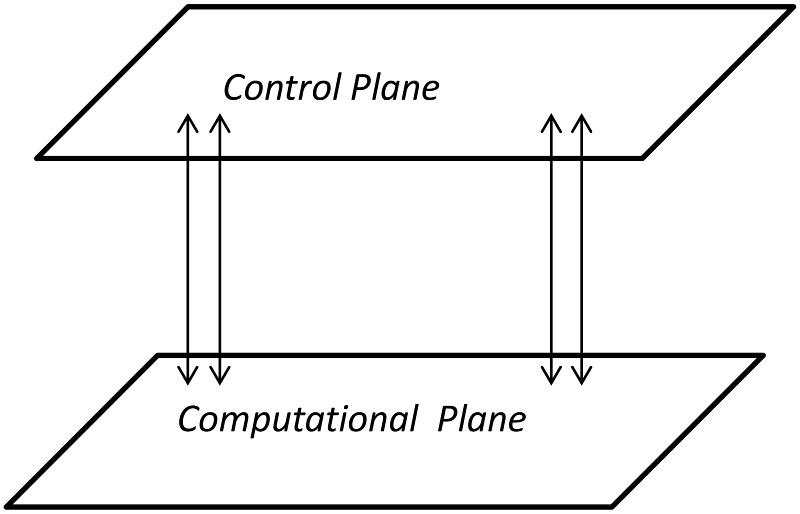


Figure 81. The proposed architecture consists mainly of a two-layer IC, with computational and compression-encryption planes stacked face to face (allowing the smallest possible distance between the layers), allowing information to flow from the computational to the compression-encryption plane (control plane) as fast as possible. Die-to-die communication is achieved using micro bumps that provide an enhanced vertical interconnection density and smaller distance, as compared to the wire-bonded method.

We first place the compression coprocessor and then the crypto coprocessor in order to ensure the highest ratio for the compression process. According to Intel, if compression is done after the encryption of data, the ratio of compression will be poor, due to the strong stochastic properties of the encrypted data [107]. Also, by compressing data first, we can assist the encryption by significantly reducing the size of the data to be encrypted. Moreover, compression increases data entropy and enhances the efficiency of encryption. Finally, it provides another layer of security to the whole structure [107].

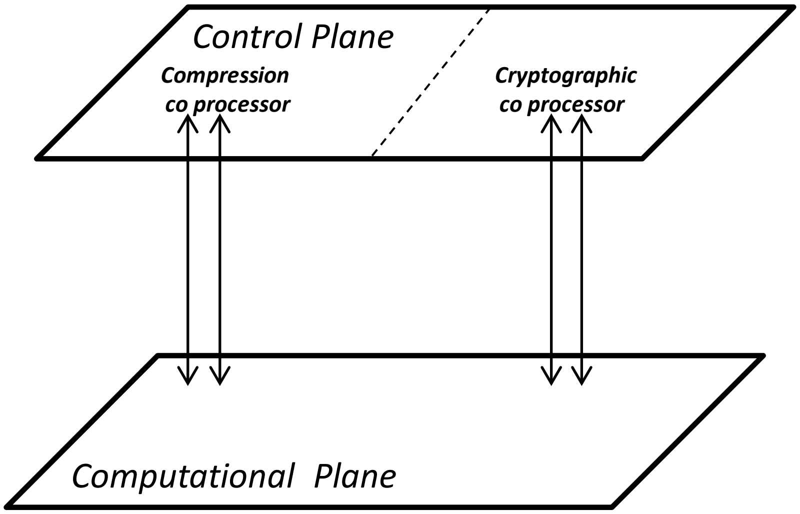


Figure 82. First, the compression coprocessor is placed in the control plane, followed by the cryptographic coprocessor.

We recommend that the control plane use the I/O capability of the computational plane rather than implement its own separate capability. Die-to-die connections will enable this sharing.

## B. OPTIONS FOR STRAWMAN-DESIGN COMPUTATIONAL PLANE

Our proposed computational plane architecture has two goals.

- Performance: comparable to other processors in the marketplace
- Traces: allow a control plane to access the information needed to generate a specific set of traces to be compressed and encrypted.

### 1. Performance

The computational plane of our proposed design has to be comparable to the most advanced processors offered in the market. For comparison, Intel is offering the i7-3930K, a processor with six cores that handles twelves threads simultaneously, running at 3.3 GHz, with a 12 MB cache and a 64-bit instruction set, in a 32 nm lithography. The memory is accessed using four memory channels, at a rate of 21 GB/s. The temperature of the case is about 162.7 $^{o}$F (72.6 $^{o}$C), and the package measures 58.5 x 51 mm [108].

### 2. Traces

Our design is also influenced by the traces we want to generate. It is impossible to track all registers in a processor, especially in hyper-threading processors where threads run in parallel, due to the giant amount of information that is processed per unit of time. We have to decide carefully what has to be monitored, e.g., by choosing registers carefully.

As an example of the complexity involved, consider memory buses. We can access all data being stored or retrieved, but data by itself has no meaning without the instructions being executed, as described in Chapter III, Trace Compression. Along with the "where to collect the traces" the other question is "when to collect it." The basic fetch–decode–execute cycle has, for example, twenty-four stages in the Pentium IV [108] including, for instance, stages that determine the length of the instruction.

Mysore et al. propose a 3D hardware approach to dynamic program analysis: "In order to ensure that the profiling hardware will be flexible enough to perform a wide variety of analysis methods, we need to capture many different signals" like memory addresses (64 bits), memory values (64 bits), program counter (64 bits), opcodes, register names, register values, cache miss, branch miss, and TLB miss. This set of signals gives an estimate of the number of inter-die vias or "direct wires that need to be accommodated for all relevant information to be passed on to an analysis engine." Based on the requirements given, they estimate that 1024 bits of profile data will be generated each cycle, which will in turn require 1024 inter-die connections." [109]

One obvious place to monitor is the control unit, collecting information from the program counter register, which holds the next instruction address, the status register that contains information such as overflows, and the instruction register that holds the next instruction to be executed. The control unit controls cores and threads using control signals. Those signals can be collected to keep track of what each core is executing. Four cores can be identified by two bits each. Data addresses are also important information that can be combined with PCs for debugging and behavior analysis.

With Intel's hyper threading, each core executes two threads [110], so adding one extra bit can identify the two threads in each core, resulting in three identification bits that help keep track of what core and thread is accessing the referenced data during a given clock cycle, making it possible to closely inspect each process.

In Chapter III, 2D Compression Hardware, we described traces consisting of program counters, branch target addresses, exception-handler target address, and data address. In general, program counters and data addresses are present in all traces that deal with program debugging and behavior analyses; therefore, program counter (PC), data address, and the special core/thread identification (CTID) are a good set of fields to consider, resulting in a 131-bit trace (64 bits PC + 64 bits Instruction + 3 bits CTID) for each core. For the purpose of this thesis and based on the previously presented architectures from the literature, we will consider traces containing a 64-bit PC and a 64-bit data address, resulting in a 128-bit trace.

This architecture requires 128 direct links (128 bits) between the computational and control planes to access (using taps) the PC (64 bits) and memory-address registers (64 bits). Those direct links will be accessible at the computational plane's face to be bonded with 128 direct links in the control plane's face, in which wires will carry signals to the trace compression hardware. No bus is used, because we are proposing a face-to-face architecture in which the distances between dies are minimal. Even the Hypertransport bus requires additional implementation costs, and is also slower than direct links.

Figure 83 presents the basic layout of the computational plane, showing the control unit of a microprocessor where PCs are stored, memory address register, cache memory, clock (used to distribute clock signals for the synchronization of computational plane with control plane and compression coprocessor with cryptographic coprocessor), I/O interface, and I/O controller to handle I/O requirements for both planes. We send the compressed and encrypted traces back to the computational plane to avoid the need for a new I/O structure in the control plane.

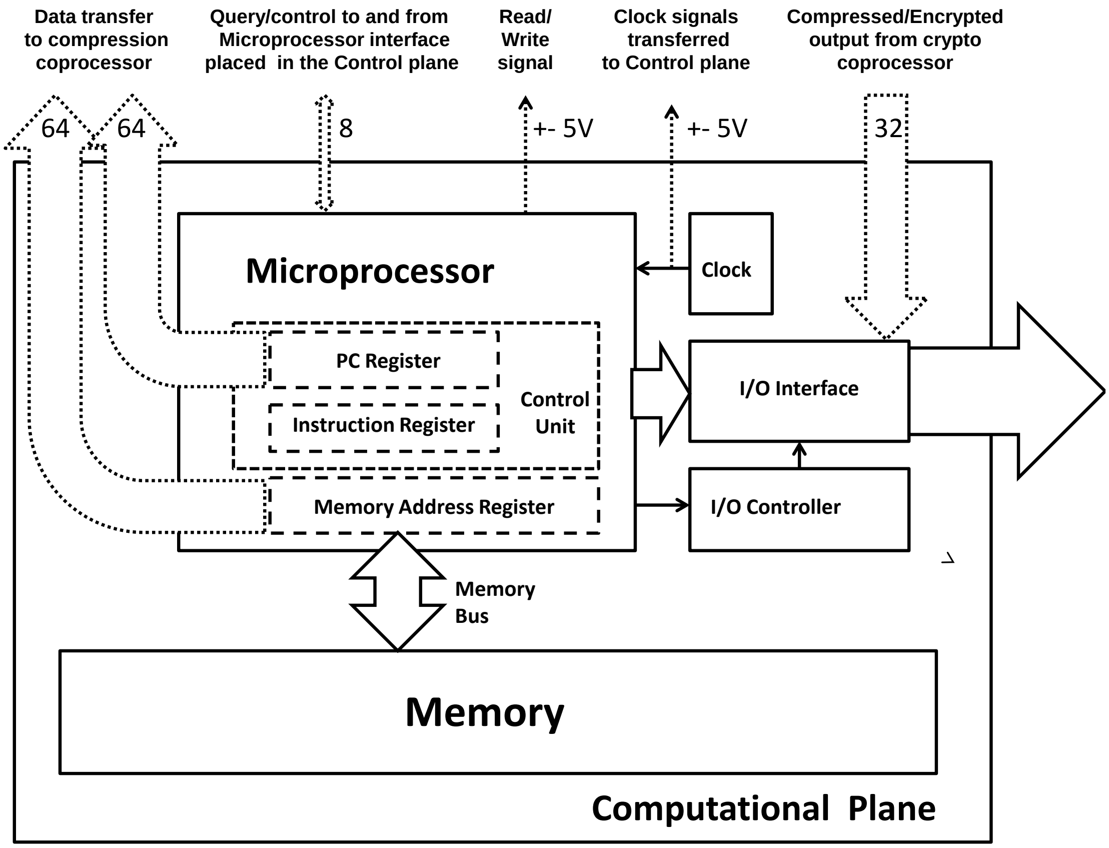

Data transfer to compression coprocessor
Query/control to and from Microprocessor interface placed in the Control plane
Read/ Write signal
Clock signals transferred to Control plane
Compressed/Encrypted output from crypto coprocessor
64
64
8
+- 5V
+- 5V
32
Microprocessor
Clock
PC Register
Instruction Register
Control Unit
I/O Interface
Memory Address Register
I/O Controller
Memory Bus
Memory
Computational Plane

Figure 83. Layout of the computational plane, showing the control unit, memory address register, cache memory, clock unit, I/O interface, and I/O controller to handle I/O requirements for both planes.

## C. CONTROL PLANE REQUIREMENTS

The control plane needs to be synchronized with the computational plane to establish a trusted communication. The main components of the control plane are the microprocessor interface described in the interface requirements, the compression coprocessor, and the crypto coprocessor.

The control plane also uses buffers, since, according to Milenkovic et al., “Internal buffers ensure that the trace compression proceeds without stalling the processor and without dropping data” [111].

## D. INTERFACE REQUIREMENTS

Not only does data need to be transferred between layers, but also clock signals and query/control signals. We now discuss how to perform this distribution and what signals will be required.

### 1. Query/Control Signals

The query and control signals are managed by a microprocessor interface in the control plane; this interface receives a clock signal, read/write signal, address/data byte, and has externally accessible registers to receive/send the signals. The proposed registers are error, status, interrupt, command, and reset.

The read/write signal consists of a simple positive/negative signal, positive meaning write, and negative meaning read. The address/data signals consist of one byte; the two most significant bits addresse the specific interface register, according to the signal purpose. The next bit defines whether the signal is for the compressor or the crypto hardware; and the next five bits are the instruction being transmitted, for a total of 32 query/control instructions for each coprocessor. Figure 84 shows the proposed interface, and Table 14 has the proposed signals.

When a write signal is received, the interface sends back an “ack” signal to the microprocessor and two coprocessors, reads the two address bits, and writes the next six bits of data to the appropriate register (error, status, interrupt, command or reset). While receiving the “ack,” the compression hardware constantly monitors those registers for a starting bit instruction that equals 0, and the crypto hardware monitors for a starting bit instruction that equals 1. The communications are synchronized by the clock signal and are shown in Figure 85 when reading signals and in Figure 86 when writing signals to the control plane.

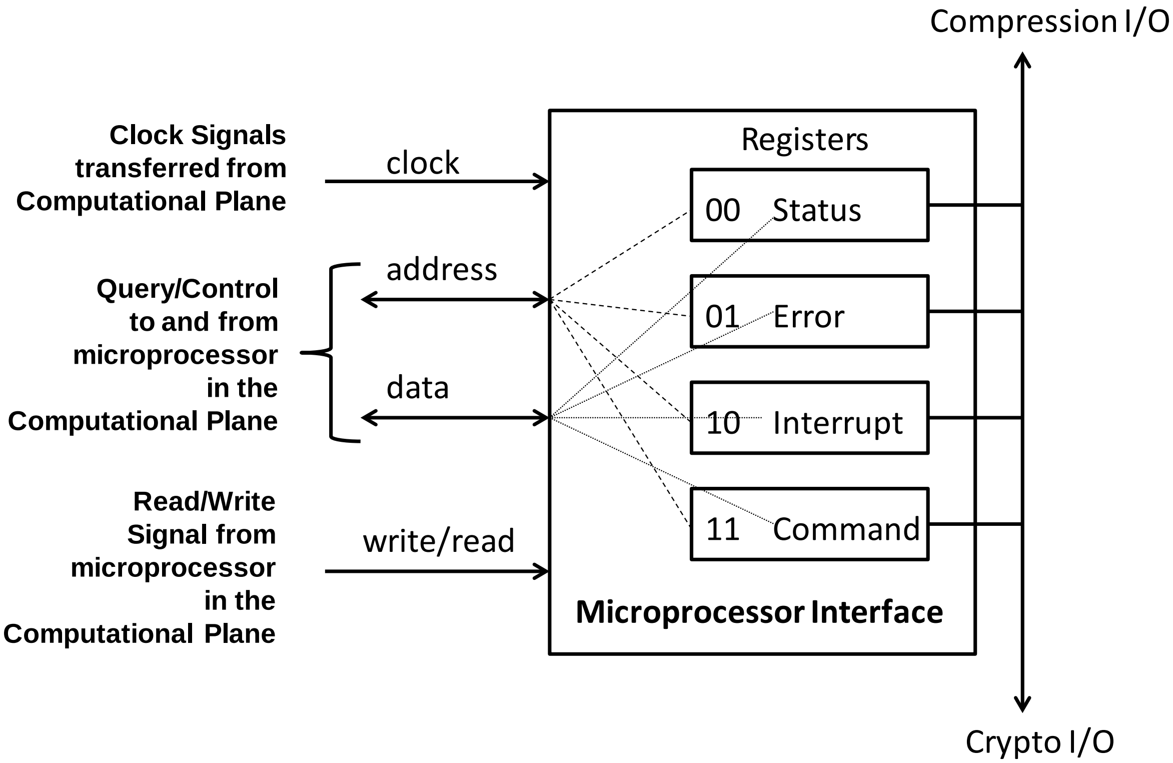

Compression I/O
Clock Signals transferred from Computational Plane
clock
Registers
00 Status
Query/Control to and from microprocessor in the Computational Plane
address
01 Error
data
10 Interrupt
Read/Write Signal from microprocessor in the Computational Plane
write/read
11 Command
Microprocessor Interface
Crypto I/O

Figure 84. The query and control signals are managed by a microprocessor interface in the control plane; this interface receives a clock signal, read/write signal, address/data byte, and has externally accessible registers to receive/send the signals. The proposed registers are error, status, interrupt, command, and reset.

| QUERY/CONTROL SIGNALS | | | | | | |
|---|---|---|---|---|---|---|
| Signal | | Description | Binary form of signal | | | Register |
| | | | register address | compr/crypto | data | |
| 1 | BUSY | While data transfer occurs | 00 | 0 | 00000 | Status |
| 2 | HOLD | HOLD command from microprocessor | 00 | 0 | 00001 | Status |
| 3 | BYPASS | No compression/decompression settled | 00 | 0 | 00010 | Status |
| 4 | ERROR | Any error from error register | 00 | 0 | 00011 | Status |
| 5 | INPUT BUFFER OVERFLOW | Input buffer overflow | 01 | 0 | 00100 | Error |
| 6 | ERROR 2 | reserved | 01 | 0 | 00101 | Error |
| 7 | ERROR 3 | reserved | 01 | 0 | 00110 | Error |
| 8 | ERROR 4 | reserved | 01 | 0 | 00111 | Error |
| 9 | ERROR INTERRUPT | Reading ERROR on status register and then error register | 10 | 0 | 01000 | Interrupt |
| 10 | DONE INTERRUPT | No input data, no error and no hold signals | 10 | 0 | 01001 | Interrupt |
| 11 | HOLD INTERRUPT | Reading HOLD on status register | 10 | 0 | 01010 | Interrupt |
| 12 | INTERRUPT 4 | reserved | 10 | | 01011 | Interrupt |
| 13 | HOLD | Hold compression | 11 | 0 | 01100 | Command |
| 14 | RESUME | Resume compression | 11 | 0 | 01101 | Command |
| 15 | START COMPRESSION | Start compression | 11 | 0 | 01110 | Command |
| 16 | START DECOMPRESSION | Start decompression | 11 | 0 | 01111 | Command |
| 17 | RESET COMPRESSION | Clear compression buffer and registers | 11 | 0 | 10000 | Command |
| 18 | MODE 1 | reserved | 11 | 0 | 10010 | Command |
| 19 | MODE 2 | reserved | 11 | 0 | 10010 | Command |
| 20 | CRYPTO READY | Manages the synchronization of the coprocessor in order to receive data. It also controls the flow of data. | 11 | 1 | 00000 | Command |
| 21 | CRYPTO SEND | 1. Manages the synchronization of the coprocessor in order to send data. It also controls the flow of data.<br>2. Used also as a halt signal. | 11 | 1 | 00001 | Command |
| 22 | AES_en | Selection of AES-128 cryptographic algorithm. | 11 | 1 | 00010 | Command |
| 23 | SHA1_en | Selection of SHA_1 cryptographic algorithm. | 11 | 1 | 00011 | Command |
| 24 | SHA512_en | Selection of SHA_512 cryptographic algorithm. | 11 | 1 | 00100 | Command |
| 25 | MODE | Selection of mode CBC or ECB for AES-128. | 11 | 1 | 00101 | Command |
| 26 | KEY | Key indication | 11 | 1 | 00111 | Command |
| 27 | RESET CRYPTO | Reset signal | 11 | 1 | 01000 | Command |

Table 14. Control / Query signals

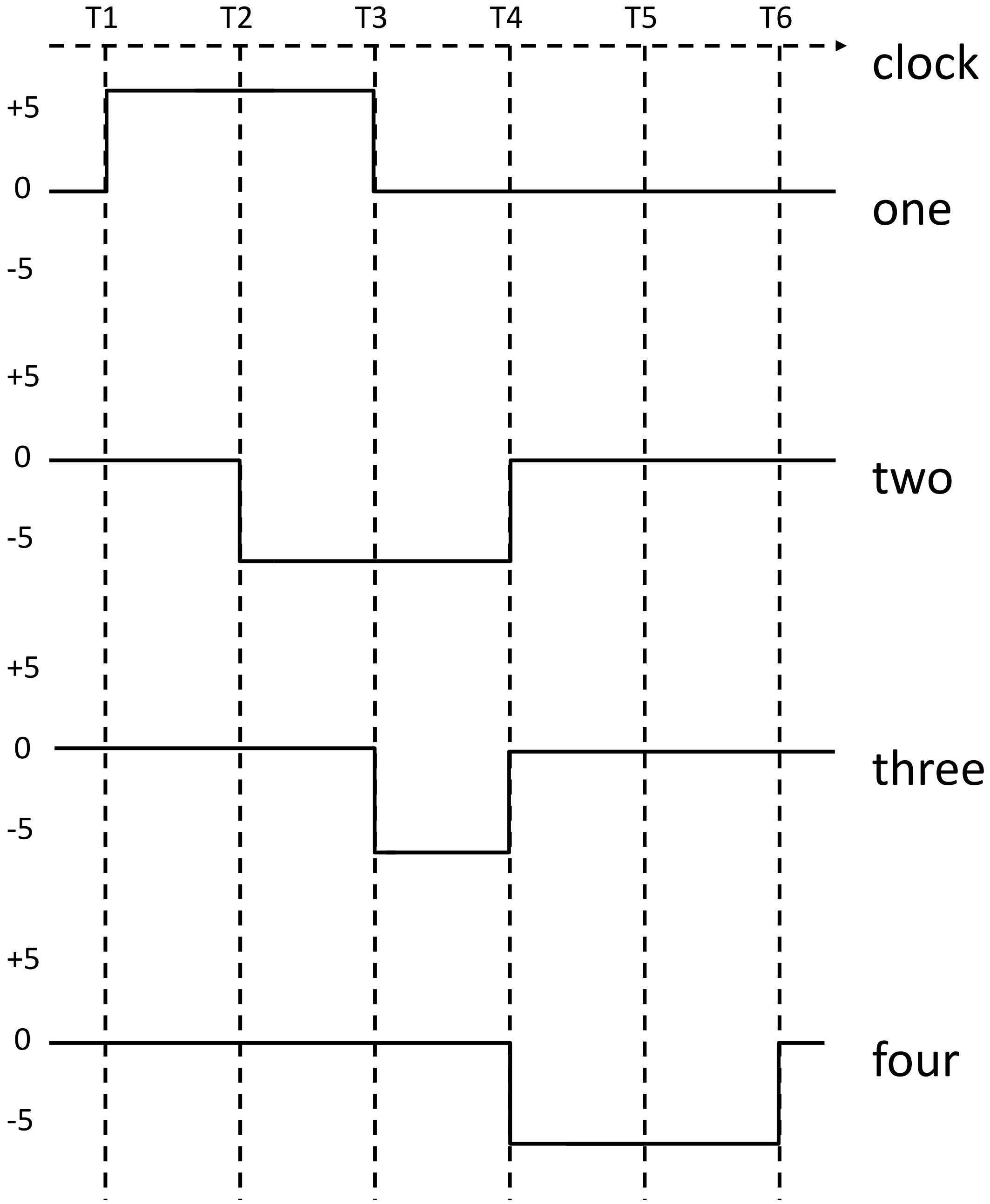


Figure 85. One: the microprocessor sends a write request (control). Two: the interface sends back an “Ack” signal to the microprocessor and the two coprocessors. Three: the interface reads the register address and opens the connection to this register. Four: data is written into the register to be read by the coprocessors. Each coprocessor will read and interpret the signal in the respective I/O interface.

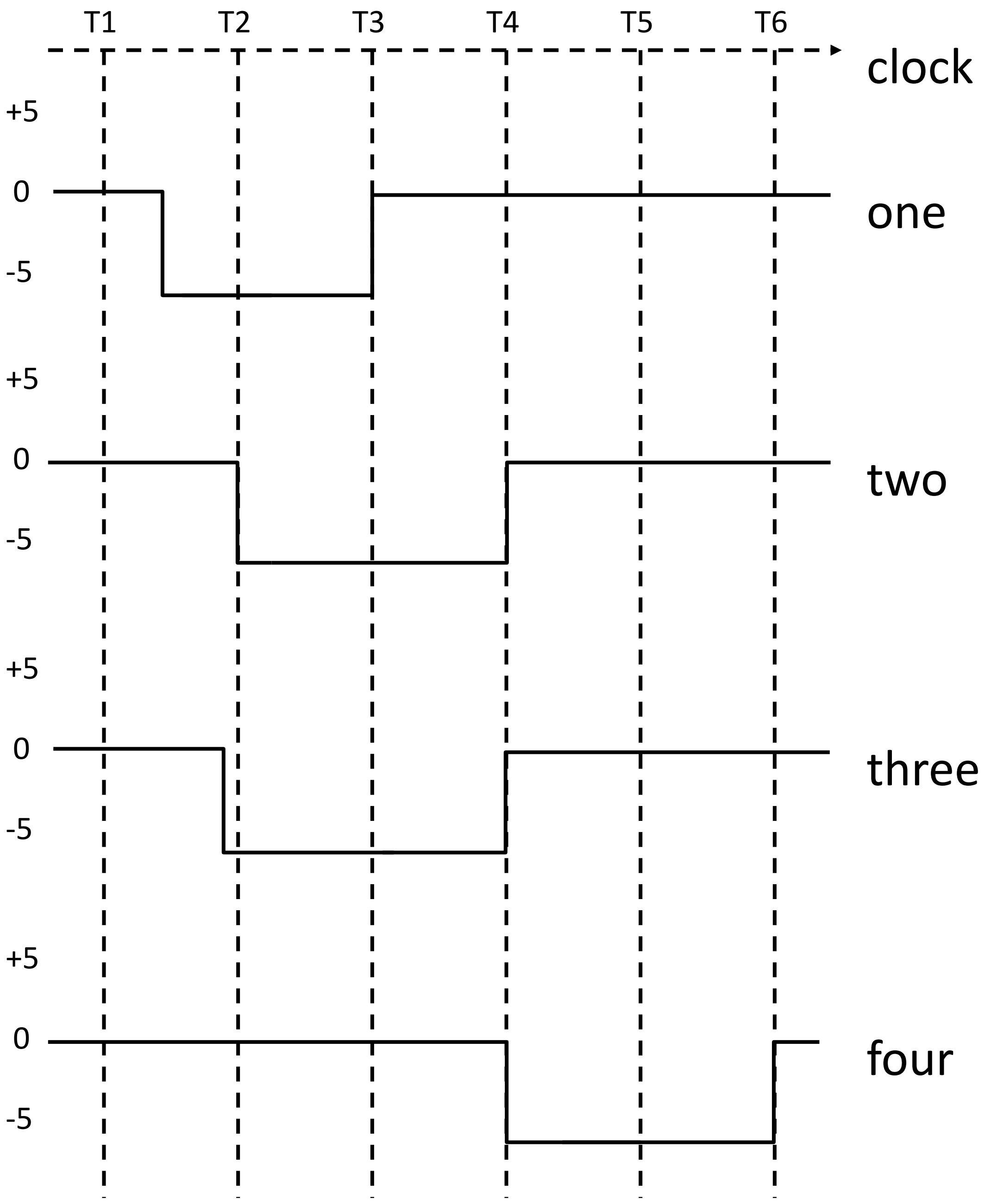


Figure 86. One: the microprocessor sends a read request (query). Two: the interface sends back an "Ack" signal to the microprocessor. Three: the interface reads the register address and opens the connection to this register. Four: data is read from the register by the microprocessor.

### 2. Clock Signals

For the synchronization of the microprocessor and the two coprocessors in the control plane, we use a clock unit with a three-level buffer clock distribution network. The buffer system provides the proper current to drive the network capacitance in conjunction with the maintaining of high quality waveform shapes (to achieve short transmission times) [112].

## E. COMPRESSION HARDWARE PARAMETERS

Compressing traces in real time inside the chip, even in a 3D architecture, requires some parameters to be considered, such as trace format, algorithms, and their required memory (so that the area fits in the control plane), while achieving high speed and a high compression ratio.

For the trace format, this thesis will consider traces recording memory access behavior, where each entry consists of program counter (PC) and the respective data address of the memory access. Both fields have 64 bits. Our decision is based on the hardware implementations from the literature described in Chapter III and the importance of these fields, as described by Milenkovic et al. “Instruction and data address traces are invaluable for quantitative evaluations of new architectures as well as for workload characterization, performance tuning, testing, and debugging” [111].

Our proposed architecture employs content addressable memories (CAM) into all memories, due to the high speed required, as described in the X-MatchPRO research, “that uses a CAM-based dictionary where multiple symbols are processed per cycle to deliver the required performance to avoid becoming a bottleneck in a system operating at a gigabit per second bandwidth” [28]. In a CAM, multiple comparisons can be made in parallel, allowing all 128 bits of trace data to be manipulated, given enough space for a CAM of this size.

We have selected two-stage compression because multiple-stage compression hardware has a better compression ratio and higher throughput than single-stage compression, as explained in Chapter III, due its ability in eliminate unnecessary data before compressing it with a general-purpose compressor like GZIP. The two stages focus on different redundancy properties in the traces: prediction methods consider context, while dictionaries do not. Therefore, we propose to combine both into one architecture.

The more specialized the compression hardware, the better the compression ratio, but some level of generality is needed for multipurpose compression hardware. Therefore, we choose to use a first stage consisting of a compressor that uses the FCM algorithm dealing with strides (DFCM). “Originally used in software-based trace compression, the finite-context method (FCM) exploits sequential locality when sets of instructions are repeatedly executed. Based upon the *n* number of previously executed instructions, a prediction of the next instruction is made.” We prefer the DFCM because “DFCM predictors are often superior to FCM predictors because they warm up faster, make better use of the hash table, and can predict values that have never been seen before. In addition to predicting long arbitrary sequences of values that repeat, DFCMs can accurately predict long arbitrary sequences of offsets (between consecutive values) that repeat” [38].

For the second stage, we propose the use of GZIP compression hardware, presented in [114]. See Figure 87. The reasons for this choice are based in the fact that “there are multiple versions of LZ compression; LZ77, LZ78 and LZW being the most common. LZ78 and LZW both generate better compression over a finite bit stream compared to LZ77. However, LZ78 and LZW both utilize static dictionaries. For this type of design, a look-up table holding the recurring symbols is required. Using a look-up table to decompress data would result in higher hardware requirements for the LZ78 and LZW algorithms. On the other hand, LZ77 utilizes a dynamic dictionary and, as a result, has a smaller impact on the memory required for decompression” [114]. GZIP is also a free algorithm, and implementations are available in the marketplace. “AHA’s current product offering includes GZIP hardware compression boards that are based on the PCI-e standard. The hardware architecture runs GZIP compression orders of magnitude faster than compression software currently available on the market, (…) the AHA367-PCIe board has four channels for a total throughput of 1.26 GByte/s” [115].

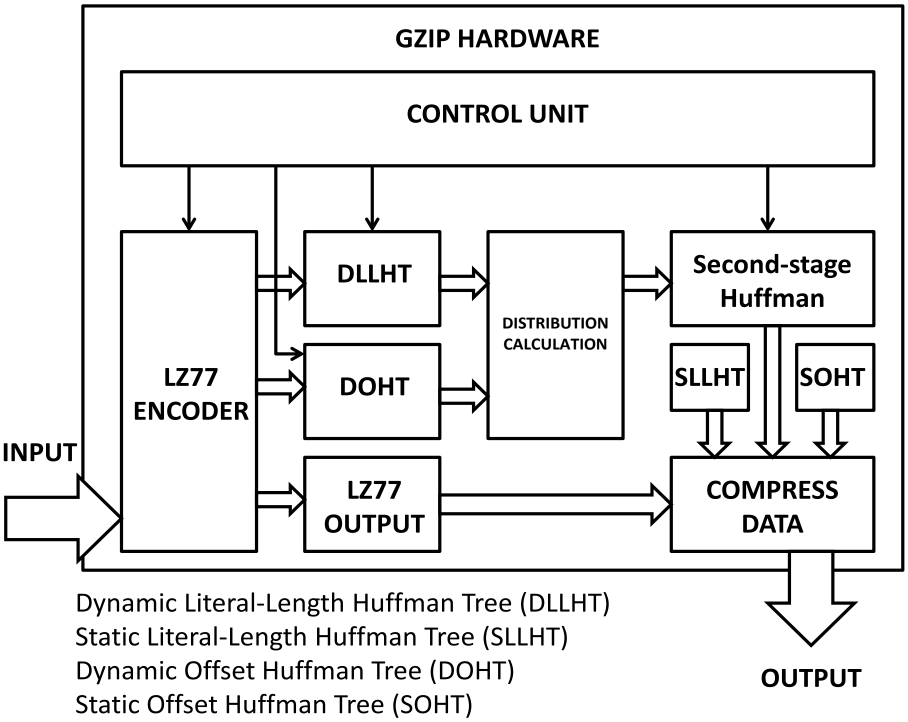

GZIP HARDWARE
CONTROL UNIT
LZ77 ENCODER
DLLHT
DOHT
LZ77 OUTPUT
DISTRIBUTION CALCULATION
Second-stage Huffman
SLLHT
SOHT
COMPRESS DATA
INPUT
OUTPUT
Dynamic Literal-Length Huffman Tree (DLLHT)
Static Literal-Length Huffman Tree (SLLHT)
Dynamic Offset Huffman Tree (DOHT)
Static Offset Huffman Tree (SOHT)

Figure 87. The stream is initially compressed using an LZ77 algorithm “which produces flags, literals, match distances and match lengths (After [114]). The literals and match lengths {0,….,285} are encoded by one Huffman tree, and the match distances {0,. . . ,29} are encoded with separate Huffman trees: the dynamic, literal-length Huffman tree (DLLHT) and the dynamic, offset Huffman tree (DOHT), or the static, literal-length Huffman tree (SLLHT) and the static, offset Huffman tree (SOHT). Once the two dynamic Huffman trees have been created, GZIP determines whether compressing the block of data with dynamic or static Huffman trees will produce a higher compression ratio. If dynamic Huffman compression is beneficial, then a representation of the DLLHT and the DOHT must occur at the beginning of the block to be able to reconstruct the Huffman trees for decompression purposes, and a third dynamic Huffman tree (second-stage Huffman) needs to be created with the alphabet {0,. . . ,18} to compress the output of DLLHT and DDHT trees. If a static Huffman tree was used, it is not necessary to output any tree since the decompressor has access to the static codes” [114].

The compression hardware has to be able to fetch 128 bits in each clock cycle, and also includes an input FIFO buffer in order to not stall the microprocessor and absorb speed variations due to prediction “warm up” times. The I/O interface receives the 6-bit control signals and writes back status signals. It also receives and distributes the single clock signal required and controls the buffer input.

Due to the requirements of our proposed crypto coprocessor architecture, we must slice (i.e., buffer) the output into 32 bits data to be encrypted. At the end of the first compression cycle, the compression coprocessor sends the “SEND” signal to the crypto coprocessor via the microprocessor interface and waits for the “READY” signal in order to start the encryption of these data.

The compression architecture is presented in Figure 88.

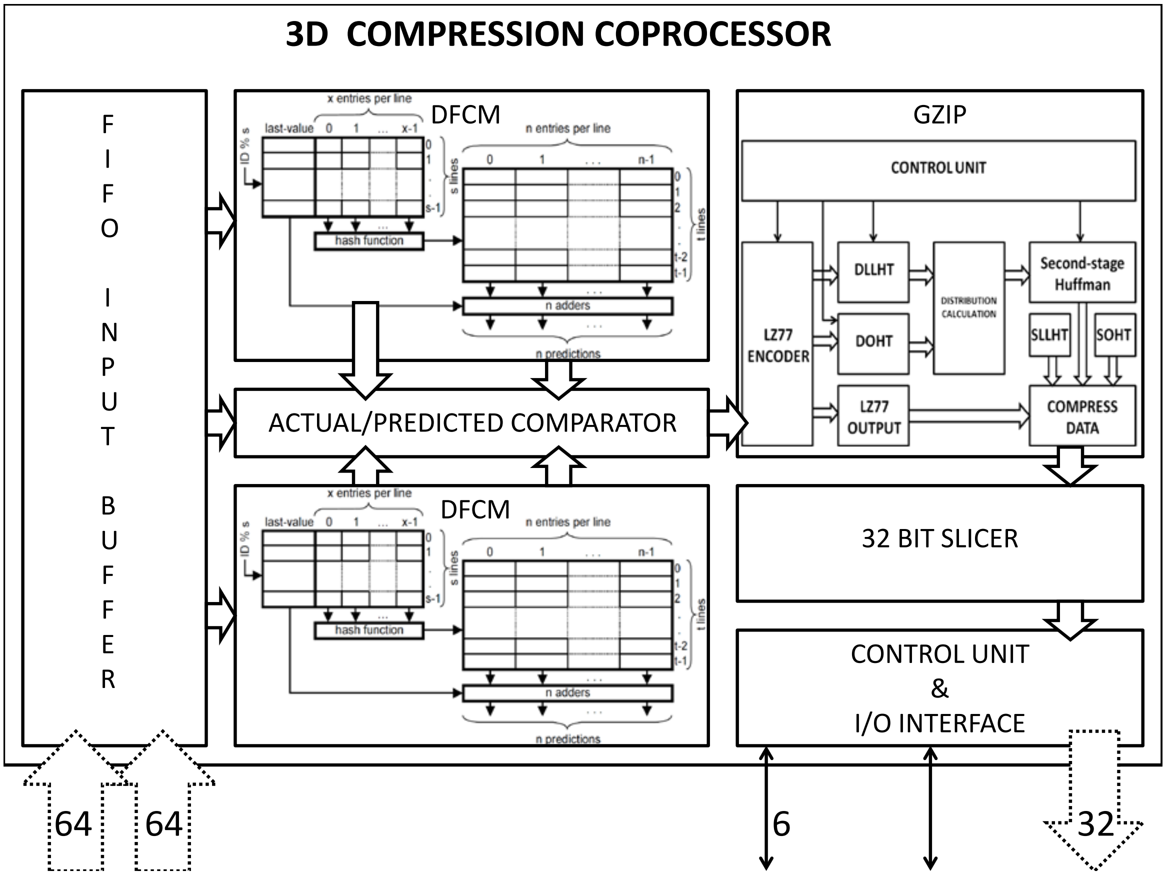


Figure 88. Traces arrive at the compression coprocessor via a FIFO input buffer. All 128 bits are received in parallel for speed reasons. They are received on the first clock signal after a "start compression" signal is received. Each trace is then sent to different DFCM compression hardware to be transformed into streams. All DFCM compressors share the same CAM, which is divided virtually among them. DFCM predicts the trace based on context and strides. Those predictions are sent to comparison hardware that compares it with the actual data being processed. If a match occurs, the address of the prediction is output to the stream. Otherwise, the uncompressed trace is output together with a miss flag. Predicted and non-predicted traces are combined into a single stream that is input to the GZIP hardware. After being compressed, traces are sliced into 32-bit chunks and sent to the crypto coprocessor.

## F. CRYPTO PARAMETERS

As described in Chapters II and IV, our crypto coprocessor is inspired by the HSSec cryptographic coprocessor [62]. The cryptographic coprocessor receives the "SEND" control signal from the compression coprocessor via the microprocessor interface in order to be ready to accept the compressed data. The cryptographic coprocessor receives the “SEND” signal at the same time as it receives the 32-bit compressed data stream. Moreover, the cryptographic coprocessor sends back to the compression coprocessor the “READY” signal via the microprocessor interface, in order to handle the synchronization with the compression coprocessor and start receiving compressed data to be encrypted. The selection of cryptographic algorithm to be used for the encryption is indicated with the control signals, ‘AES_en” and “MODE” for the AES-128 algorithm or “SHA1_en” or “SHA512_en” for the SHA1 or SHA512 algorithms. These signals in addition to the “KEY” and “RESET” signals, are transmitted from the microprocessor through its interface to the HSSec cryptographic coprocessor. The 32-bit output is compressed, and the final data stream is handled by the I/O controller and two I/O interfaces. One of the I/O interfaces is placed in the computational plane, and the other is placed in the cryptographic coprocessor in the control plane.

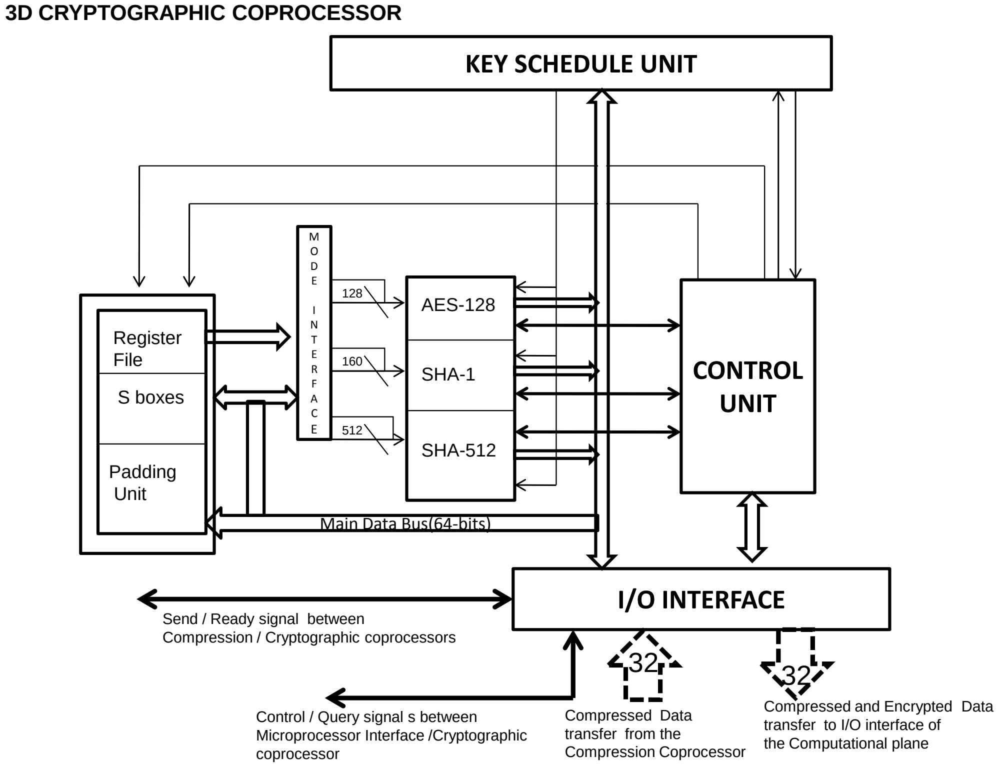

3D CRYPTOGRAPHIC COPROCESSOR
KEY SCHEDULE UNIT
Register File
S boxes
Padding Unit
MODE INTERFACE
128
160
512
AES-128
SHA-1
SHA-512
CONTROL UNIT
Main Data Bus(64-bits)
I/O INTERFACE
Send / Ready signal between Compression / Cryptographic coprocessors
Control / Query signal s between Microprocessor Interface /Cryptographic coprocessor
32
Compressed Data transfer from the Compression Coprocessor
32
Compressed and Encrypted Data transfer to I/O interface of the Computational plane

Figure 89. The cryptographic coprocessor architecture utilized for the 3DIC (After [62]).

The control unit manages data processing and communication with the compression coprocessor and microprocessor interface. Cryptographic primitives (AES-128, SHA-1, and SHA-512) are arranged in a parallel orientation and utilize a common 64-bit global data bus. The key scheduler block is used for key expansion and generating message schedules. The memory block consists of a register file, padding unit, and S-boxes. The mode interface is responsible for modifying the input to the cryptographic primitives. The key scheduler performs the RotWord and SubWord transformations described in Chapter IV. The key scheduler also provides constants needed by the hash functions: SHA-1 uses a sequence of eighty constant 32-bit words, and SHA-512 uses a sequence of eighty constant 64-bit words.

The overall architecture is presented in Figure 89.

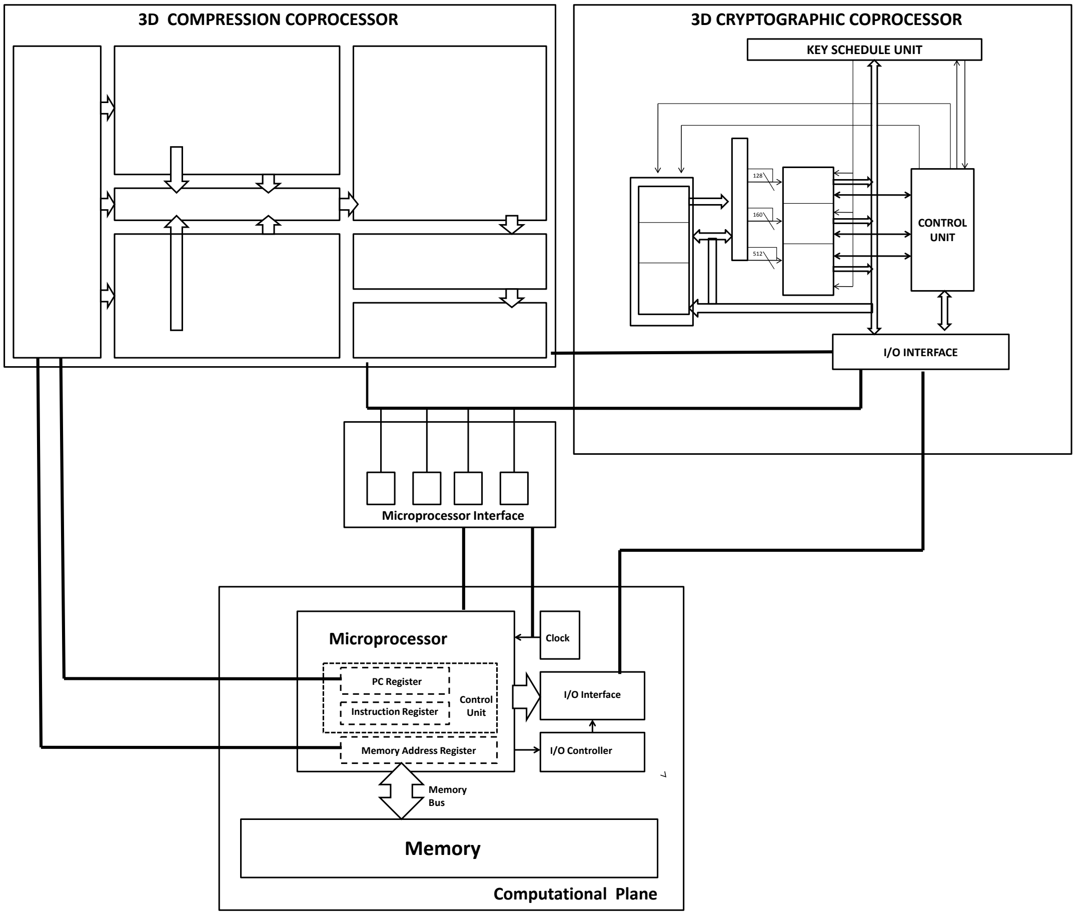


Figure 90. Block diagram showing the integration of the computation plane, microprocessor interface, compression unit, and cryptographic unit into a full system

# VII. RESULTS SUMMARY

## A. SUMMARY

In the previous chapters we explored the architecture and the design of 3DICs. We presented the advantages and challenges of this emerging technology. Moreover, we explored the various compression and cryptographic features, as well as related algorithms and their efficiency and performance. In Chapter VI, we proposed a two-layer IC, with a computational plane and a compression-encryption plane stacked in a face-to-face fashion, allowing the information flow from the computational plane, where a general-purpose processor is placed, to the control plane, where compression-encryption circuitry resides. The die-to-die communication is achieved using micro-bumps, which provide enhanced vertical interconnect density and small distance as compared to the wire-bonded method. The main application that is supported by our proposed 3DIC is one that performs real-time trace collection, compressing the trace and then encrypting the compressed trace (data), protecting it from interception.

## B. CONCLUSIONS

The 3DIC containing a general-purpose processor and coprocessors for compression and encryption provides the following advantages:

- The average interconnection wire length is reduced significantly, as compared to a traditional two-dimensional (2D) design; therefore, the overall performance of the system is enhanced.
- Without a bus width limitation for uncompressed data, the proposed architecture is able to collect more data per unit time using direct links, compress and encrypt it, and then use a common off-chip bus.

- Due to the decrease in the average interconnection length, which consequently leads to a reduction in total wiring, we can achieve less power consumption for our proposed 3DIC.According to Huffmire et al., placing a cryptographic coprocessor in the control plane and decoupling its operation from the computational plane is an example of a secure alternate service (SAS), which provides “a trustworthy enhancement or alternative to the service provided in the computational plane” [3].
- By placing in the control plane both the compression coprocessor and the crypto coprocessor (with the output of the compression coprocessor connected to the input of the crypto coprocessor), we can enable a higher compression ratio than in the opposite configuration. According to Elbaz et al., if compression is done after the encryption of data, the ratio of compression will be poor, due to the strong stochastic properties of the encrypted data. Also, by compressing data first, we can enhance the encryption performance by significantly reducing the size of the data to be encrypted. Moreover, compression increases data entropy and therefore enhances the efficiency of encryption. Finally, it provides another layer of security to the whole structure [107].
- Another advantage of a 3D approach is that the computational plane can be manufactured in an untrusted foundry and the control plane can be manufactured in a trusted foundry. The two dies are joined in a trusted facility. Dual use of the computational plane provides economic benefits, but requires small changes to the plane. Fortunately, these changes are amortized over all custom 3D designs that use the same modified computation plane.
- The reduction in power consumption can allow the use of longer encryption key lengths (e.g., AES-192 or AES-128). Therefore, we can enhance the security of the encrypted data without suffering greater power consumption.

## C. ANALYTICAL RESULTS SUMMARY

Although we are not building a hardware prototype, for simulation purposes we used software to evaluate different algorithms to confirm or refute our architecture choices. The software collects traces of Linux program execution, applies the DFCM algorithm in the first compression stage, and applies the GZIP algorithm in the second compression stage. We measure compression performance in terms of compression ratio.

To perform this experiment, we used a trace file capturing the memory access behavior of five Linux applications, generated using the Pin dynamic binary instrumentation tool. We also used TCGen, which, given the trace file's format, generates working C code and can apply four different algorithms and vary each algorithm's configurations. In our analysis, we used a total of 112 configurations. TCGen also allows the use of a second compression stage. In our analysis, we used only GZIP because of the advantages explained in Chapters III and VI.

The traces consist three fields: instruction counter, program counter, and data address. We used only two: the program counter and data address, for the reasons explained in Chapter VI. The traces were collected from the execution of five Linux programs—GIMP, Open Office, Opera, Firefox, and Mozilla—using the Pin dynamic binary instrumentation tool developed by Intel and freely available to the public.

The algorithms compared are the differential finite-context-method (*DFCMx[n]),* finite-context-method predictor (*FCMx[n]*), stride predictor (*ST[n]*), and last *n* values predictor (*LV[n]*), all described in Chapter III. We vary *n* and *x* for each algorithm from 1 to 7, making all combinations as shown in Table 14.

The results are expressed in terms of percentage of good predictions, i.e., the number of times an algorithm can correctly predict the next input. Although this is not the only factor that affects compression (see Chapter III), the percentage of good predictions is strongly related to it. All algorithms run independently, and the value *unpredictable* is the number of traces that no algorithm could predict.

Our objective is to find the algorithm that most often yields a correct prediction (and therefore has a better compression ratio) and use it to guide our selection of compression hardware for our proposed design

| ALGORITHM | n | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| DFCM7[n] | | | | | | | |
| DFCM6[n] | | | | | | | |
| DFCM5[n] | | | | | | | |
| DFCM4[n] | | | | | | | |
| DFCM3[n] | | | | | | | |
| DFCM2[n] | | | | | | | |
| DFCM1[n] | | | | | | | |
| FCM7[n] | | | | | | | |
| FCM6[n] | | | | | | | |
| FCM5[n] | | | | | | | |
| FCM4[n] | | | | | | | |
| FCM3[n] | | | | | | | |
| FCM2[n] | | | | | | | |
| FCM1[n] | | | | | | | |
| ST[n] | | | | | | | |
| LV[n] | | | | | | | |

Table 15. Table used to collect the percentage of good predictions made by different algorithms: differential finite-context-method (*DFCMx[n])*, finite-context-method predictor (*FCMx[n]*), stride predictor (*ST[n]*), and last *n* values predictor (*LV[n]*), varying *x* and *n* from 1 to 7.

The method in Table 15 was applied to each of the two fields of each of the five program traces. Then the results from the same fields in different program traces were combined and the mean was calculated, resulting in one performance graph for each field.

| program/trace field | First step | Second Step | Result |
|---|---|---|---|
| GIMP field one | Run 112 algorithms | Compute the mean of each one of the 112 algorithms/configurations on field one | Output a graphical representation |
| Open Office field one | Run 112 algorithms | | |
| Opera field one | Run 112 algorithms | | |
| Firefox field one | Run 112 algorithms | | |
| Mozilla field one | Run 112 algorithms | | |
| GIMP field two | Run 112 algorithms | Compute the mean of each one of the 112 algorithms/configurations on field two | Output a graphical representation |
| Open Office field two | Run 112 algorithms | | |
| Opera field two | Run 112 algorithms | | |
| Firefox field two | Run 112 algorithms | | |
| Mozilla field two | Run 112 algorithms | | |

Table 16. The methodology of trace compression and analysis was applied to each of the two fields of each of the five program traces. Then the results from the same fields of different program traces were combined and the mean was calculated, resulting in one performance graph for each field.

The resulting graph in Figure 93 for field one (program counter) shows that DFCM1[n] is the best algorithm for this specific field, especially when *n* is greater than five, with 70% good predictions. If memory is a concern, the best algorithm is DFCM1[n] with *n* equal to one, with 53.6% good predictions. This result validates our design choice of DFCM for the compression hardware, with a performance of, at most, 70% good predictions. The same is not true of field two, for which the best algorithm is FCM1[n]. The best result occurs when *n* is equal to four, with 46% good predictions. This result shows that in general, data addresses do not have a fixed stride and the differential property of DFCM is not contributing. Therefore, we recommend the use of an FCM algorithm for data addresses, with a performance of, at most, 46% good predictions.

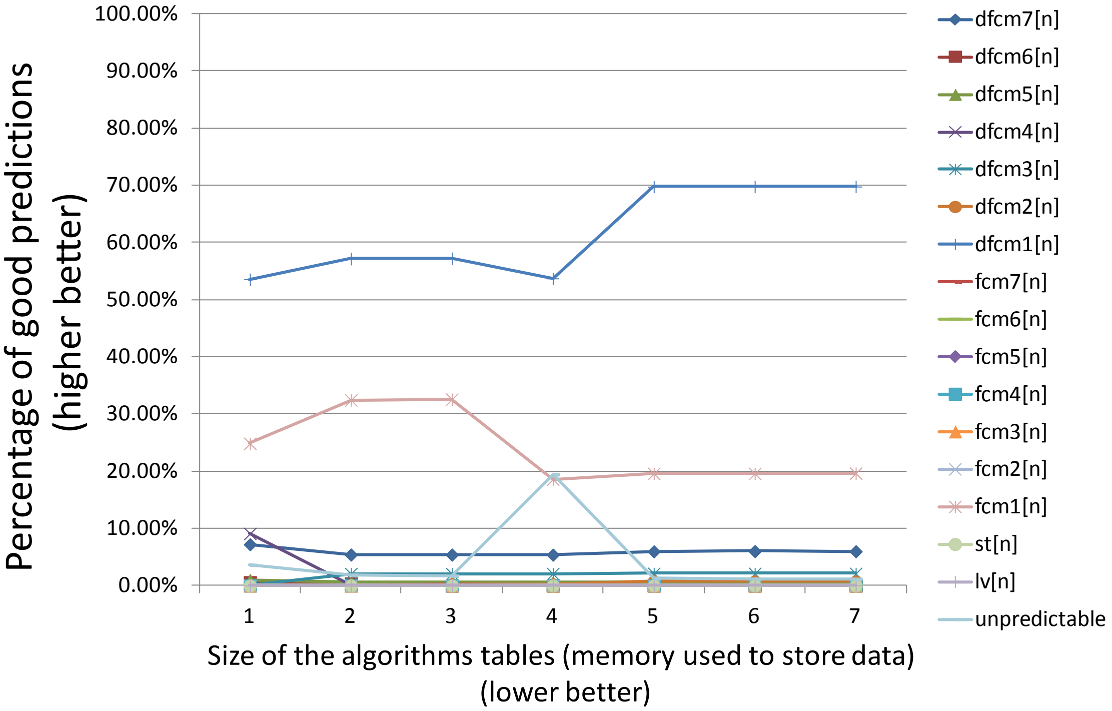


Figure 91. Program Counter: The DFCM1[n] algorithm is the best algorithm for this specific field, especially when *n* is greater than five, with 70% good predictions. If memory is a concern, the best algorithm is DFCM1[*n*] with *n* equal to one, with 53.6% good predictions. This result confirms our design choice of DFCM, with a performance of, at most, 70% good predictions.

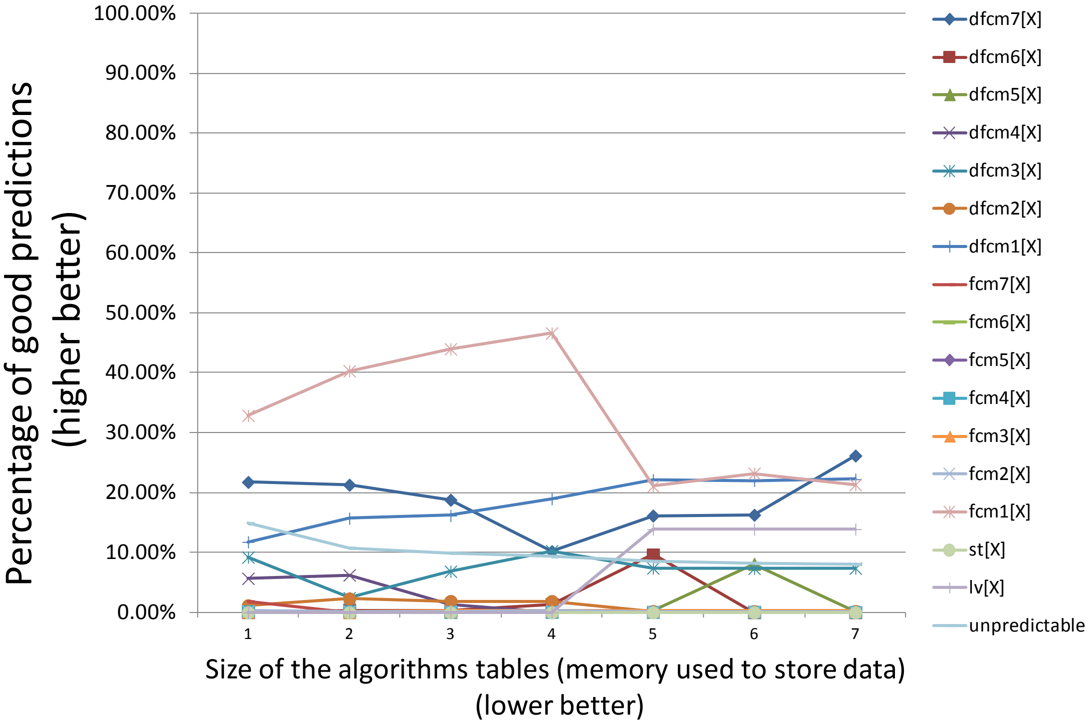


Figure 92. Data Address: The FCM1[*n*] is the best algorithm for this specific field. The best result is when *n* is equal to four, with 46% good predictions. This result shows that data addresses do not have fixed stride and the differential property of DFCM is not contributing. Therefore, we recommend using a FCM algorithm for data addresses, with a performance of, at most, 46% good predictions.

The resulting output of this phase for each program trace was then sent to the second compression stage, based on GZIP (see Figure 92), and the final compression ratio was compared against a single compression stage, consisting only of GZIP (see Table 16), to determine and quantify whether our two-stage proposal is more efficient than a single-stage one.

| program / trace field | First step | Second Step | Result |
|---|---|---|---|
| GIMP field one | Run best algorithm plus GZIP | Compare compression ratio of the two-stage architecture with one-stage architecture and also obtain the mean compression ratio | Output a graphical representation |
| | Run GZIP | | |
| Open Office field one | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Opera field one | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Firefox field one | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Mozilla field One | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| GIMP field Two | Run best algorithm plus GZIP | Compare compression ratio of the two-stage architecture with one-stage architecture and obtain the mean compression ratio | Output a graphical representation |
| | Run GZIP | | |
| Open Office field Two | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Opera field Two | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Firefox field Two | Run best algorithm plus GZIP | | |
| | Run GZIP | | |
| Mozilla field Two | Run best algorithm plus GZIP | | |
| | Run GZIP | | |

Table 17. The resulting output of the first phase for each program trace was then sent to the second compression stage, based on GZIP, and the final compression ratio was compared against a single compression stage, consisting only of GZIP.

The mean compression ratio for the five program traces on field one shows that our two-stage proposal (DFCM + GZIP) has a slight advantage over a single GZIP stage.

Although this performance may not justify the cost of the architecture, using two-stage compression can speed up the process by pre-compressing traces with a trace-specialized tool before sending it to a general-purpose compressor; also, the power and memory required for the general-purpose compression can be reduced, due to the pre-compressing unit. For the second field, GZIP alone performs better, with a compression ratio of 33:1, compared with 25:1 for our proposed design. The two graphs show that the performance of the first stage has to be around 70% (as in the first field) to effectively contribute to the overall compression ratio.

The first compression stage's algorithm has to be carefully chosen and optimized for the specific trace being compressed; otherwise, it will be better to use a single-stage approach. A way to deal with this problem is a decision mechanism that observes the first stage's prediction performance; it chooses to use either one- or two-stage compression, based on the percentage of good predictions.

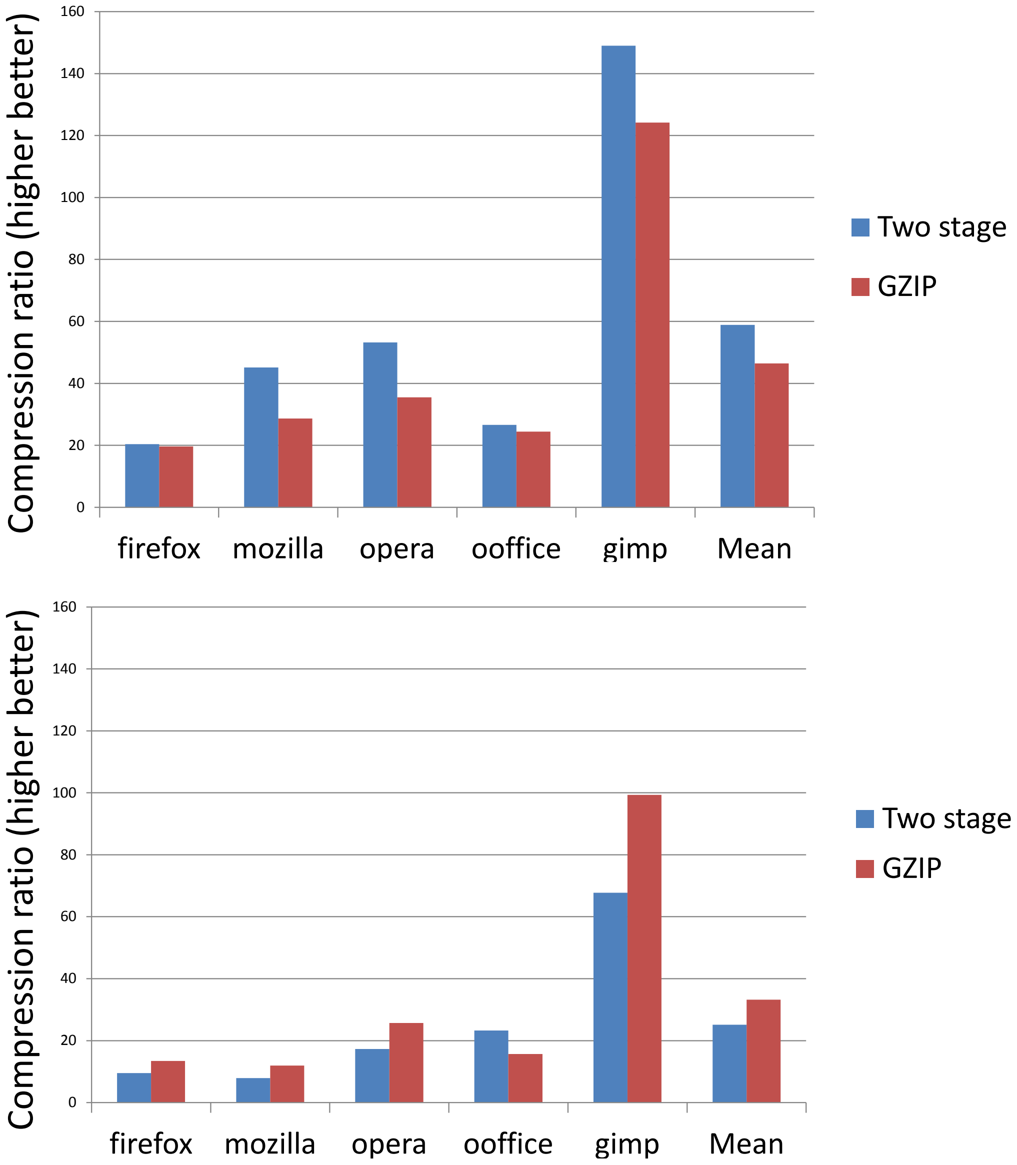


Figure 93. (Upper) Program Counter: The mean compression ratio for the five program traces, showing that our two-stage proposal (DFCM + GZIP) has a slight advantage over a single GZIP stage. (Lower) Data Address: The poor percentage of good predictions in field two reflects in the poor compression ratio in field two for our proposed design. The first stage's algorithm has to be carefully chosen in order to achieve a better compression ratio.

## D. FUTURE WORK

In this thesis, we have proposed strawman architecture for a two-die, three-dimensional processor with compression and crypto coprocessors for trace collection. Future work in this area will include:

- Additional trace studies: debugging, profiling, and security all have a unique set of trace requirements.
- Determining the speed of data generated to be compressed/encrypted: based on the traces collected and the speed of their generation, we can compute the amount of data per time unit that needs to be compressed and encrypted.
- Defining the best algorithm for each trace in the first stage compression: as shown in this thesis, the first compression step requires a very specific and optimized algorithm to achieve a better compression ratio and throughput. Analysis has to be conducted for each specific trace.
- Implementing a hardware simulation: The simulator is essential to define throughput and real performance and provides a better understanding of the memory and area required.
- Determining the throughput of the device: given the amount of data to be processed and the memory requirements determined by the simulation, the throughput needs to be enough to process this data.
- Determine the number/area of TSVs, not only for data but also power: this is an important 3D design decision. Not only does the number of TSVs need to be calculated, but also the special 3D package requirements need to be determined.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX A DESCRIPTION OF SHA-1 BASED ON THE FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 180-2

## A. OPERATIONS

In order to describe the algorithm, we have to use the following operations on strings of 32 bits [17,18]:

1. $X \wedge Y$ bitwise “and”
2. $X \vee Y$ bitwise “or”
3. $X \oplus Y$ bitwise addition mod 2
4. $\neg X$ flips “0 to 1” and ” 1 to 0 “
5. $X + Y$ addition of X and Y mod $2^{32}$ where X,Y are integers mod $2^{32}$
6. The rotate left (circular left shift) operation, $ROTL^{n}(x) = (x << n) \vee (x >> w - n).$, where x is a w-bit word, and *n* is an integer with 0 ≤ *n* < w. $ROTL^{n}(x)$ is equivalent to a circular shift (rotation) of x by *n* positions to the left.

7. Functions.

   SHA-1 uses a sequence of logical functions, $f_0, f_1, \ldots, f_{79}$. Each function $f_t$, where 0 ≤ *t* < 79, operates on three 32-bit words, *x*, *y*, and *z*, and produces a 32-bit word as an output. The function $f_t$ (*x*, *y*, *z*) is defined as follows [17]:

$$f_{t(x,y,z)=} \begin{cases} \text{Ch(x, y, z)=(x} \wedge \text{y)} \oplus (\neg\text{x} \wedge \text{z)} & 0 \le \text{t} \le 19 \\ \text{Parity(x,y,z)=x} \oplus \text{y} \oplus \text{z} & 20 \le \text{t} \le 39 \\ \\ \text{Maj(x,y,z)=(x} \wedge \text{y)} \oplus \text{(x} \wedge \text{z)} \oplus \text{(y} \wedge \text{z)} & 40 \le \text{t} \le 59 \\ \text{Parity(x,y,z)=x} \oplus \text{y} \oplus \text{z} & 60 \le \text{t} \le 79 \end{cases} \quad (1)$$

8. Constants.

   SHA-1 uses a sequence of eighty constant 32-bit words, $K_0, K_1, \ldots, K_{79}$, which are given by [17]:

$$K_{t=}\begin{cases} \text{5a827999} & 0 \le \text{t} \le 19 \\ \text{6ed9eba1} & 20 \le \text{t} \le 39 \\ & \\ \text{8f1bbcdc} & 40 \le \text{t} \le 59 \\ \text{ca62c1d6} & 60 \le \text{t} \le 79 \end{cases} \qquad (2)$$

## B. PREPROCESSING

Preprocessing takes place before the initiation of the hash-computation process. This preprocessing stage consists of three steps: 1) padding the message, *M*, 2) parsing the padded message into blocks, and 3) determining the initial hash value, $H^{(0)}$ [17].

### 1. Padding the Message

Message M must be padded [17] before the hash-computation stage. Padding the message M ensures that the padded message is a multiple of 512 bits.

### 2. Parsing the Padded Message

During this stage [17], the padded message *M* is parsed into *N m*-bit blocks before the hash computation stage. The padded message is parsed into *N* 512-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 512 bits of the input block can be represented by sixteen 32-bit words, the first 32 bits of the message block *i* are denoted as $M_0^{(i)}$, the next 32 bits are denoted as $M_1^{(i)}$, and so on until $M_{15}^{(i)}$ [17].

### 3. Setting the Initial Hash Value ($H^{(0)}$)

Before the hash-computation stage [17], the initial hash value, $H^{(0)}$, should be determined. The size and number of words in $H^{(0)}$ are related to the message digest size. The initial hash value, $H^{(0)}$, consists of the following five 32-bit words, in hex [17]:

$H_0^{(0)}$ = 67452301

$H_1^{(0)}$ = efcdab89

$H_2^{(0)}$ = 98badcfe

$H_3^{(0)}$ = 10325476

$H_4^{(0)}$ = c3d2e1f0

## C. SHA-1

SHA-1 may be used to hash a message, *M*, having a length of $\ell$ bits, where $0 \leq l \leq 2^{64}$. The algorithm uses 1) a message schedule of eighty 32-bit words, 2) five working variables of 32 bits each, and 3) a hash value of five 32-bit words. The final result of SHA-1 is a 160-bit message digest. The words of the message schedule are labeled $W_{0,}\ W_{1,}\ ......, W_{79.}$ . The five working variables are labeled ***a***, ***b***, ***c***, ***d***, and ***e***. The words of the hash value are labeled $H_{0,}{}^{(i)}H_{1,}{}^{(i)}......, H_4{}^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-1 also uses a single temporary word, *T*. [17]

### 1. SHA-1 Preprocessing

1. Pad the message, *M*.
2. Parse the padded message into *N* 512-bit message blocks, $M^{(1)}, M^{(2)}, \ldots,$ $M^{(N)}$.
3. Set the initial hash value, $H^{(0)}$.

### 2. SHA-1 Hash Computation

The SHA-1 hash computation uses functions and constants. Addition (+) is performed modulo $2^{32}$. After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}, \ldots,$ $M^{(N)}$, is processed in order, using the following steps: [17]

```
For i=1 to N:
{
    1. Prepare the message schedule, {$W_t$}:
        $W_{t=} \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ ROTL^1( W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \le t \le 79 \end{cases}$

    2. Initialize the five working variables, a, b, c, d, and e, with the
       $(i-1)^{st}$ hash value:

            $a = H_0^{(i-1)}$
            $b = H_1^{(i-1)}$
            $c = H_2^{(i-1)}$
            $d = H_3^{(i-1)}$
            $e = H_4^{(i-1)}$

    3. For $t$ = 0 to 79:
       {
            $T = ROTL^5(a) + f_t(b,c,d) + e + K_t + W_t$
            $e = d$
            $d = c$
            $c = ROTL^{30}(b)$
            $b = a$
            $a = T$
       }
    4. Compute the $i^{th}$ intermediate hash value $H^{(i)}$:
            $H_0^{(i)} = a + H_0^{(i-1)}$
            $H_1^{(i)} = b + H_1^{(i-1)}$
            $H_2^{(i)} = c + H_2^{(i-1)}$
            $H_3^{(i)} = d + H_3^{(i-1)}$
            $H_4^{(i)} = e + H_4^{(i-1)}$
}
```

After repeating steps one through four a total of $N$ times (i.e., after processing $M^{(N)}$), the resulting 160-bit message digest of the message, $M$, is: $H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)}$.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX B DESCRIPTION OF SHA-512 *BASED ON THE FEDERAL INFORMATION PROCESSING STAND*ARDS PUBLICATION 180-2

## A. OPERATIONS

In order to describe the algorithm, we have to use the following operations on strings of 64 bits [17,18].

1. $X \wedge Y$ bitwise "and"
2. $X \vee Y$ bitwise "or"
3. $X \oplus Y$ bitwise addition mod 2
4. $\neg X$ flips "0 to 1" and " 1 to 0 "
5. $X + Y$ addition of X and Y mod $2^{64}$ where X,Y are integers mod $2^{64}$

6. The right shift operation, $SHR^{n}(x) = x >> n$ , where x is a w-bit word, and *n* is an integer, with 0 ≤ *n* < w.
7. The rotate right operation, $ROTR^{(n)}(x) = (x >> n) \vee (x << w - n)$, where x is a w-bit word, and *n* is an integer, with 0 ≤ *n* < w. $ROTR^{(n)}(x)$ is equivalent to a circular shift (rotation) of x by *n* positions to the right.
8. Functions.

SHA-512 use six logical functions, where each function operates on 64-bit words, which are represented as *x*, *y*, and *z*. The result of each function is a new 64-bit word [17].

$$\begin{aligned} &\text{Ch(x, y, z)=(x} \wedge \text{y)} \oplus (\neg\text{x} \wedge \text{z)} \\ &\text{Parity(x,y,z)=x} \oplus \text{y} \oplus \text{z} \end{aligned} \qquad (1),(2)$$

$$\begin{aligned} &\sum_{0}^{\{512\}}(\chi) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \\ &\sum_{1}^{\{512\}}(\chi) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \end{aligned} \qquad (3),(4)$$

$$\sigma_0^{\{512\}}(\chi) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$$
$$\sigma_1^{\{512\}}(\chi) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \qquad (5),(6)$$

9. Constants

SHA-512 uses a sequence of eighty constant 64-bit words, $K_0^{\{512\}}, K_1^{\{512\}}, .........., K_{79}^{\{512\}}$. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. A detailed list of these constant values in hex format is available in [17].

## B. PREPROCESSING

Preprocessing takes place before hash computation begins. This preprocessing consists of three steps: padding the message, *M*, parsing the padded message into blocks, and setting the initial hash value, $H^{(0)}$ [16].

### 1. Padding the Message

The message, *M*, shall be padded before hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 1024 bits. Suppose that the length of the message, *M*, is $\ell$ bits. Append the bit “1” to the end of the message, followed by *k* zero bits, where *k* is the smallest, non-negative solution to the equation $\ell$ +1+k=896mod1024. Then append the 128-bit block that is equal to the number expressed using a binary representation [17].

### 2. Parsing the Padded Message

After a message has been padded, it must be parsed into *N* *m*-bit blocks before the hash computation can begin. The padded message is parsed into *N* 1024-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 1024 bits of the input block may be expressed as sixteen 64-bit words, the first 64 bits of message block *i* are denoted as $M_0^{(i)}$, the next 64 bits are denoted as $M_1^{(i)}$, and so on, up to $M_{15}^{(i)}$ [17].

### 3. Setting the Initial Hash Value $\left(H^{(0)}\right)$

Before hash computation begins for each of the secure hash algorithms, the initial hash value, $H^{(0)}$, must be set. The size and number of words in $H^{(0)}$ depends on the message digest size. The initial hash value, $H^{(0)}$, shall consist of the following eight 64-bit words, in hex [17]:

$H_0^{(0)} = \text{cbbb9d5dc1059ed8}$
$H_1^{(0)} = \text{629a292a367cd507}$
$H_2^{(0)} = \text{9159015a3070dd17}$
$H_3^{(0)} = \text{152fecd8f70e5939}$
$H_4^{(0)} = \text{67332667ffc00b31}$
$H_5^{(0)} = \text{8eb44a8768581511}$
$H_6^{(0)} = \text{db0c2e0d64f98fa7}$
$H_7^{(0)} = \text{47b5481dbefa4fa4}$

## C. SHA-512

SHA-512 may be used to hash a message, *M*, having a length of l bits, where $0 \leq l < 2^{128}$. The algorithm uses 1) a message schedule of eighty 64-bit words, 2) eight working variables of 64 bits each, and 3) a hash value of eight 64-bit words. The final result of SHA-512 is a 512-bit message digest. The words of the message schedule are labeled *W*0, *W*1,…, *W*79. The eight working variables are labeled ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, .......H_7^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-512 also uses two temporary words, *T*1 and *T*2 [17].

### 1. SHA-512 Preprocessing

1. Pad the message, *M*.
2. Parse the padded message into *N* 512-bit message blocks, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$.

3. Set the initial hash value, $H^{(0)}$.

## 2. SHA-512 Hash Computation

The SHA-512 hash computation uses functions and constants. Addition (+) is performed modulo $2^{64}$. After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}$ ,…, $M^{(N)}$, is processed in order, using the following steps:[17]

For i=1 to *N*:

{

1. Prepare the message schedule,{ $W_t$ }:

$$W_{t=} \begin{cases} M_t^{(i)} & 0 \le \mathrm{t} \le 15 \\ \sigma_1^{\{512\}}(\mathrm{W}_{t-2}) + \mathrm{W}_{t-7} + \sigma_0^{\{512\}}(\mathrm{W}_{t-15}) + \mathrm{W}_{t-16} & 16 \le \mathrm{t} \le 79 \end{cases}$$

2. Initialize the eight working variables, ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***, with the $(i-1)^{st}$ hash value:

$$a = H_0^{(i-1)}$$
$$b = H_1^{(i-1)}$$
$$c = H_2^{(i-1)}$$
$$d = H_3^{(i-1)}$$
$$e = H_4^{(i-1)}$$
$$f = H_5^{(i-1)}$$
$$g = H_6^{(i-1)}$$
$$h = H_7^{(i-1)}$$

3. For t=0 to 79:

{

```
                $T_1 = h + \sum_{1}^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t$
                $T_2 = h + \sum_{0}^{\{512\}}(a) + Maj(a, b, c)$
                $h = g$
                $g = f$
                $f = e$
                $e = d + T_1$
                $d = c$
                $c = b$
                $b = a$
                $a = T_1 + T_2$
            }
        4. Compute the $(i)^{th}$ intermediate hash value $H^{(i)}$:
                $H_0^{(i)} = a + H_0^{(i-1)}$
                $H_1^{(i)} = b + H_1^{(i-1)}$
                $H_2^{(i)} = c + H_2^{(i-1)}$
                $H_3^{(i)} = d + H_3^{(i-1)}$
                $H_4^{(i)} = e + H_4^{(i-1)}$
                $H_5^{(i)} = f + H_5^{(i-1)}$
                $H_6^{(i)} = g + H_6^{(i-1)}$
                $H_7^{(i)} = h + H_7^{(i-1)}$
    }
```

After repeating steps one through four a total of *N* times (i.e., after processing $M^{(N)}$), the resulting 512-bit message digest of the message, *M*, is:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES

[1] T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine, “Trustworthy System Security through 3D Integrated Hardware,” *Proceedings of the 2008 IEEE International Workshop on Hardware—Oriented Security and Trust (HOST),* Anaheim, CA, June 2008.

[2] T. Huffmire, T. Levin, C. E. Irvine, T.D Nguyen, J. Valamehr, R. Kastner, and T. Sherwood, “High-Assurance System Support through 3D Integration,” NPS Technical Report NPS-CS-07-016, November 2007.

[3] T. Huffmire, T. Levin, M. Bilzor, C. E. Irvine, J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner,”Hardware trust implications of 3D integration,” October 2010 WESS '10: *Proceedings of the 5th Workshop on Embedded Systems Security*.

[4] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, T. Levin, “Hardware assistance for trustworthy systems through 3D integration,” December 2010 ACSAC '10: *Proceedings of the 26th Annual Computer Security Applications Conference,* ACM New York, NY, 2010.

[5] A. Vasudevan, Qu. Ning Qu, A. Perrig, "XTRec: Secure Real-Time Execution Trace Recording on Commodity Platforms," System Sciences (HICSS), 2011 44th Hawaii International Conference on System Sciences, vol., no., pp.1–10, 4–7 Jan. 2011.

[6] G. H. Loh, Y. Xie, B. Black ,”Processor Design in Three-Dimensional Die-Stacking Technologies,” In *IEEE Micro*, vol. 27(3), pp. 31–48, May–June, 2007.

[7] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” In *Proceedings Of The 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '05). ACM, New York, NY, USA, 190–200.

[8] D. Salomon, *A Concise Introduction to Data Compression,* 1938

[9] D. Salomon, G. Motta and D. C. O. *N*. Bryant, *Handbook of Data Compression,* 2009.

[10] H.-H. Sean Lee, ISCA-35 “Tutorial 3D-IC Microarchitecture,” School of Electrical and Computer Engineering Georgia Institute of Technology, ISCA 2008.

[11] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir. “Design and Management of 3D Chip Multiprocessors Using Network-in-Memory,” In *33rd International Symposium on Computer Architecture (ISCA)*, pages 130–141, 2006

[12] S. Das et al. Technology, “Performance, and Computer Aided Design of Three-Dimensional Integrated Circuits,” In *Proc. International Symposium on Physical Design*, 2004.

[13] S. Jung et al, “The Revolutionary and Truly 3Dimentional 25F2 SRAM Technology with the Smallest S3 Cell, 0.16um2 and SSTFF for Ultra High Density SRAM,” In *VLSI Technology Digest of Technical Papers*, 2004.

[14] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, N. Vijaykrishnan, M. S. Yousif, C. R. Das, “A Novel Dimensionally-Decomposed Router for On-Chip Communication in 3D Architectures,” ISCA’07, June 9–13, 2007, San Diego, California, USA.

[15] Kung, S.David, X. Yuan, "Guest Editors' Introduction: Opportunities and Challenges of 3D Integration," Design & Test of Computers, *IEEE*, vol. 26, no.5, pp. 4–5, Sept.–Oct. 2009.

[16] J. Joyner, P. Zarkesh-Ha, and J. Meindl, “A stochastic global net-length distribution for a three-dimensional system-on-achip (3D-SoC),” In *Proc. 14th Annual IEEE International ASIC/SOC Conference*, Sept. 2001.

[17] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, “Die stacking (3D) microarchitecture,” In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, FL, December 2006.

[18] A. J. Menezes, P. C. Van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptograpy*, 1997.

[19] L. C. Washington and W. Trappe, *Introduction to Cryptography: With Coding Theory*, 2002.

[20] D. E. Robling Denning, *Cryptography and Data Security*, 1982.

[21] B. Schneier, *Applied Cryptography*, second edition.

[22] NIST, “Introduction to Public Key Technology and the Federal PKI Infrastructure,” SP 800-32.

[23] C. S. Walls, “Survey of Security Processors and Accelerators,” ECE 798 Research Paper.

[24] T. Bell and D. Kulp, (1993), Longest-match string searching for ziv-lempel compression. Software: Practice and Experience, 23: 757–771. doi: 10.1002/spe.4380230705.

[25] I.-C. K. Cheng, J. T. Coffey, and T. N. Mudge, “Analysis of branch prediction via data compression,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[26] T. Bell, *“Better OPM/L Text Compression*,*” Communications, IEEE Transactions on 34(12)*, pp. 1176–1182. 1986.

[27] M. B. Lin, J. F. Lee and G. E. Jan, “A lossless data compression and decompression algorithm and its hardware architecture,*” Very Large Scale Integration (VLSI) System”s, IEEE Transactions on 14(9)*, pp. 925–936, 2006.

[28] J. L. Núñez and S. Jones, “Gbit/s lossless data compression hardware,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 11(3)*, pp. 499–510. 2003.

[29] V. Uzelac, A. Milenković, M. Burtscher and M. Milenković, “Real-time unobtrusive program execution trace compression using branch predictor events,” *Presented at Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2010,

[30] M. Milenkovic, A. Milenkovic and M. Burtscher, “Algorithms and hardware structures for unobtrusive real-time compression of instruction and data address traces,” *Presented at Data Compression Conference*, 2007.

[31] A. Milenković and M. Milenković. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 17(1)*, pp. 2. 2007.

[32] A. Milenkovic and M. Milenkovic, “Exploiting streams in instruction and data address trace compression,” *Presented at Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop*, 2003.

[33] M. Burtscher, “VPC3: A fast and effective trace-compression algorithm,” *Presented at ACM SIGMETRICS Performance Evaluation Review*, 2004.

[34] M. Burtscher and M. Jeeradit, “Compressing extended program traces using value predictors,” *Presented at Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, 2003.

[35] M. Burtscher and B. G. Zorn. Exploring last n value prediction. Presented at Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on. 1999.

[36] B. Goeman, H. Vandierendonck and K. De Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. Presented at High-Performance Computer Architecture, 2001. HPCA. the Seventh International Symposium on. 2001,

[37] M. Burtscher. An improved index function for (D) FCM predictors. *ACM SIGARCH Computer Architecture News 30(3)*, pp. 19-24. 2002.

[38] M. Burtscher and N. B. Sam, "Automatic generation of high-performance trace compressors," *Presented at Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, 2005.

[39] M. Burtscher, "TCgen 2.0: A tool to automatically generate lossless trace compressors" *ACM SIGARCH Computer Architecture News 34(3)*, pp. 1–8, 2006.

[40] IBM, (November 1994). ALDC 1-40S-M Data Sheet. [Online]. Available: http://icwic.cn/icwic/data/pdf/cd/cd075/Data%20Compression,%20 Encryption/a/117440.pdf.

[41] Cotech AHA corporation, (2005). AHA 3580 Product brief. [Online]. Available: http://www.datasheetarchive.com.

[42] AHA Products Group, [Online]. Available: http://www.aha.com

[43] T. Chilimbi, R. Jones and B. Zorn, "Designing a trace format for heap allocation events," *Presented at ACM SIGPLAN Notices*, 2000.

[44] A. Agarwal, R. L. Sites and M. Horowitz, "ATUM: A new technique for capturing address traces using microcode," *Presented at ACM SIGARCH Computer Architecture News*, 1986.

[45] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys (CSUR) 29(2)*, pp. 128–170, 1997.

[46] J. R. Larus, "Efficient program tracing," *Computer 26(5)*, pp. 52-61, 1993.

[47] C. F. Kao, S. M. Huang and I. J. Huang, "A hardware approach to real-time program trace compression for embedded processors," *Circuits and Systems I: Regular Papers, IEEE Transactions on 54(3)*, pp. 530–543, 2007.

[48] R. Anderson, M. Bond, J. Clulow, S. Skorobogatov, "Cryptographic processors—a survey".

[49] IBM, IBM 4764 Model 001 PCI-X, Cryptographic Coprocessor. [Online]. Available:http://www-03.ibm.com/security/cryptocards/pdfs/4764-001_PCIX_Data_Sheet.pdf

[50] S. W. Smith, S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks 31 _1999*. 831–860.

[51] B. Yee ,J. D. Tygar, "Secure Coprocessors in Electronic Commerce Applications," In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, July 1995, pp. 155–170.

[52] IBM,Cryptocards,[Online].Available:http://www6.software.ibm.com/software/cryptocards/IBM%204758.

[53] IBM,SpecificationSheet.[Online].Available:http://www03.ibm.com/secure cryptocards.

[54] S. Kent, "Protecting Externally Supplied Software in Small Computers," Ph.D. dissertation, Massachusetts Institute of Technology, 1980.

[55] S. R. White and L. Comerford. "ABYSS: A trusted architecture for software protection," In *Proc. of the IEEE Symposium on Security and Privacy*, 1987.

[56] E. R. Palmer. "An introduction to Citadel, a secure crypto coprocessor for workstations," Technical Report RC18373, IBM T. J. Watson Research Center, 1992.

[57] S. R. White, S. H. Weingart, W. C. Arnold, and E. R. Palmer. "Introduction to the Citadel Architecture: Security in Physically Exposed Environments." Technical Report RC16672, IBM T. J. Watson Research Center, 1991.

[58] J. D. Tygar and B. Yee. Dyad: "A system for using physically secure coprocessors," In *Proc. of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Networked Multimedia Environment*, April 1993.

[59] J. D. Tygar, B. Yee, and A. Spector. "Strongbox: Support for self-securing programs," In *Proc. of the (First) USENIX UNIX Security Workshop*, August 1988.

[60] B. Yee, "Using Secure Coprocessors." Ph.D. dissertation, Carnegie Mellon Univerisity, May 1994.

[61] B. Yee and J. D. Tygar. “Secure coprocessors in electronic commerce applications.” In *the 1st USENIX Workshop on Electronic Commerce*, July 1995.

[62] A.P. Kakarountas, H. Michail, C.E. Goutis, C. Efstathiou, “Implementation of HSSec: a High-Speed Cryptographic Coprocessor,” 2007.

[63] Douglas R. Stinson, *Cryptography Theory and Practice*, Third edition, Chapman & Hall /CRC.

[64] NIST, SHA Standard, National Institute of Standards and Technology (NIST), Secure Hash Standard, FIPS PUB 180-3, [Online]. Available:www.itl.nist.gov/fipspubs/fip180-3.htm, 2008.

[65] Wadde Trappe, Lawrence C.Washington, *Introduction to Cryptography with coding theory*, Second Edition, Pearson Prentice Hall.

[66] NIST, National Institute of Standards and Technology, standard 197, *The Advanced Encryption Standard (AES)*, 2001.

[67] A. Brokalakis, A.P. Kakarountas, C.E. Goutis, “A High- Throughput Area Efficient FPGA Implementation of AES-128 Encryption,” in *Proc. of IEEE 2005 International Workshop on Signal Processing Systems (SiPS’05)*, Athens, Greece, pp. 116-121, Nov. 2–4, 2005.

[68] C.P Su, T.F Lin, C.T Huang, and C.W Wu, “A High-Throughput Low-Cost AES Processor,” *IEEE Communications Magazine*, December 2003.

[69] S. Mangard, M. Aigner, and S. Dominikus, “A Highly Regular and Scalable AES Hardware Architecture,” *IEEE Transactions On Computers*, Vol. 52, No. 4, April 2003.

[70] National Institute of Standards and Technology, Special Publication 800-38A, *Recommendation for Block Cipher Modes of Operation Methods and Techniques,*2001 edition.

[71] P. Stanica, Block Ciphers, Excerpt from “Cryptographic Boolean Functions and Applications" by T.W. Cusick & P. Stanica.

[72] S.W. Smith, *Trusted Computing Platforms:Design and Applications*, Springer.

[73] B. Schneier, D. Whiting, “A Performance Comparison of the Five AES Finalists.”

[74] Crypto++5.6.0Benchmarks.[Online].Available:http://www.cryptopp.com/benchmarks.html

[75] P. Dhawan, “Performance Comparison: Security Design Choices.” [Online].Ava*ilable:http://msdn.microsoft.com/enus/library/ms978415(d=printer).aspx*

[76] NetOverview. [Online].Available:http://msdn.microsoft.com/library/zw4w595w.aspx.

[77] B. Gladman, SHA1, SHA2, HMAC and Key Derivation in C. [Online]. Available:http://gladman.plushost.co.uk/oldsite/cryptography_technology/sha/index.php

[78] A.K.A Tamimi. “Performance Analysis of Data Encryption Algorithms,” [Online]. Available:http://www.cs.wustl.edu/~jain/cse56706/encryption_perf.htm

[79] S. Hirani, “Energy Consumption of Encryption Schemes in Wireless Devices*,”* M.S thesis, University of Pittsburgh, 2003.

[80] N. R. Potlapally, S. Ravi, A.Raghunathan and N. K. Jha, “Analyzing the Energy Consumption of Security Protocols,” *ISLPED’03,* August 25–27, 2003.

[81] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon, “Demystifying 3D ICs: The Pros and Cons of Going Vertical,” *IEEE Design & Test of Computers*, 22(6):498–510, 2005.

[82] A. Young. “Perspectives on 3D-IC Technology,” Presentation at the 2nd Annual Conference on 3D Architectures for Semiconductor Integration and Packaging, June 2005.

[83] A.W Topol et al, “Three dimensional integrated circuits,” *IBM Journal of R&D* ,Volume 50 ,Number 4/5 ,2006

[84] Euronymous, “3D Integration: A Revolution in Design” [Online]. Available:http://realworldtech.com/page.cfm?ArticleID=RWT050207213.

[85] J. Cong, Jie Wei, and Yan Zhang. “A thermal-driven floorplanning algorithm for 3D ICs,” In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '04). IEEE Computer Society, Washington, DC, USA, 306–313.

[86] M. Healy, M. Vittes, M. Ekpanyapong, C. Ballapuram, S.K. Lim, H.-Hsin S. Lee, and G.H. Loh. “Microarchitectural floorplanning under performance and thermal tradeoff”. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings* (DATE '06). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1288–1293.

[87] P. Falkenstern, Y. Xie, Y.-W. Chang and Y. Wang. “Three-dimensional integrated circuits (3D IC) floorplan and power/ground network co-synthesis”. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference* (ASPDAC '10). IEEE Press, Piscataway, NJ, USA, 169–174.

[88] C.C. Liu et al., ‘‘Bridging the Processor-Memory Performance Gap with 3D IC Technology,’’ *IEEE Design &Tes*t, vol. 22, no. 6, 2005, pp. 556–564.

[89] T. Kgil et al., ‘‘PicoServer: Using 3D Stacking Technology to Enable a Compact Energy EfficientChip Multiprocessor,’’ *Proc. 12th Int’l Conf. Architectural Support for Programming Languagesand Operating Systems.*

[90] H. Sun, J. Liu, Anigundi, R.S., N. Zheng, J.-Q. Lu, Rose, K., Tong Zhang, "3D DRAM Design and Application to 3D Multicore Systems," *Design & Test of Computers, IEEE*, vol.26, no.5, pp.36–47, Sept.–Oct. 2009.

[91] R.E. Matick and S.E. Schuster, “Logic-Based eDRAM: Origins and Rationale for Use,” IBM J. Research and Development, vol. 49, no. 1, Jan. 2005, pp.145–165.

[92] G.H Loh, "3D-Stacked Memory Architectures for Multi-core Processors," *Computer Architecture, 2008. ISCA '08. 35th International Symposium on* , vol., no., pp.453–464, 21–25 June 2008.

[93] M. Ghosh, H.-H.S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, vol., no., pp.134–145, 1–5 Dec. 2007 doi: 10.1109/MICRO.2007.13.

[94] K. Puttaswamy, G. H. Loh “Thermal Analysis of a 3D Die-Stacked High-Performance Microprocessor,” *GLSVLSI’06*, April 30–May 2, 2006, Philadelphia, PA, USA.

[95] J. Cong and Y. Zhang. “Thermal via planning for 3D ICs.” In *ICCAD ’05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, pages 745–752, Washington, DC, USA, 2005. IEEE Computer Society.

[96] S. S. Sapatnekar, “Addressing thermal and power delivery bottlenecks in 3D circuits,” ASP-DAC '09: *Proceedings of the 2009 Asia and South Pacific Design Automation Conference* ,January 2009.

[97] H. Hua, C. Mineo, K. Schoenfliess, A. Sule, S. Melamed, R. Jenkal, W.R. Davis, "Exploring compromises among timing, power and temperature in

three-dimensional integrated circuits," *Design Automation Conference, 2006 43rd ACM/IEEE*, vol., no., pp.997–1002, 0–0 0.

[98] B. Goplen and S. Sapatnekar." Efficient thermal placement of standard cells in 3D ICs using a force directed approach." In *International Conference on Computer Aided Design (ICCAD)*, pages 86–89, 2003.

[99] W.-L. Hung, G. Link, Y. Xie, N. Vijaykrishnan, and M. J. Irwin. "Interconnect and Thermal-aware Floorplanning for 3D Microprocessors." In *7th International Symposium on Quality Electronic Design (ISQED)*, pages 98–104, 2006.

[100] Z. Li, X. Hong, Q. Zhou,S. Zeng, J. Bian, H. Yang,V. Pitchumani, C.-K. Cheng, "Integrating Dynamic Thermal Via Planning with 3D Floorplanning Algorithm," *ISPD'06*, April 9–12, 2006.

[101] G. L. Loh, B. Agrawal, N. Srivastava, L. Sheng-Chih ,T Sherwood, K. Banerjee, "A thermally-aware performance analysis of vertically integrated (3D) processor-memory hierarchy," *Design Automation Conference, 2006 43rd ACM/IEEE*, vol., no., pp.991–996, 0–0 0.

[102] H. S. Lee, K. Chakrabarty, "Test Challenges for 3D Integrated Circuits," *Design & Test of Computers, IEEE*, vol.26, no.5, pp.26–35, Sept–Oct. 2009.

[103] A. Grochowski, D. Bhattacharya, T. R. Viswanathan, K. Laker, "Integrated circuit testing for quality assurance in manufacturing: history, current status, and future trends," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on* , vol.44, no.8, pp.610–633, Aug 1997 doi: 10.1109/82.618036.

[104] X. Wu , P. Falkenstern , K. Chakrabarty , Y. Xie, "Scan-chain design and optimization for three-dimensional integrated circuits," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, v.5 n.2, p.1–26, July 2009.

[105] P.Emma, E. Kursun, "Opportunities and Challenges for 3D Systems and Their Design," *Design & Test of Computers, IEEE*, vol.PP, no.99, pp.1, 0 doi: 10.1109/MDT.2009.98.

[106] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon,"Demystifying 3D ICs: The Pros and Cons of Going Vertical," *IEEE Design & Test of Computers*, 22(6):498–510, 2005.

[107] R.Elbaz, L.Torres, G.Sassatelli, P.Guillemin, C.Anguille, M.Bardouillet, C.Buatois, J.B. Rigaud, "Hardware Engines for Bus Encryption: a Survey of Existing Techniques," 2005.

[108] Intel, Intel® Core™ i7-3930K Processor Specifications. 2012. [Online]. Available: http://www.intel.com.

[109] S. Mysore, B. Agrawal, N.Srivastava, S.-C. Lin, K. Banerjee/Timothy Sherwood "Introspective 3D Chips".

[110] L. Null and J. Lobur, *Computer organization and architecture*.

[111] M. Milenkovic, A. Milenkovic and M. Burtscher "Algorithms and Hardware Structures for unobtrusive Real-Time Compression of Instruction and Data Address Traces".

[112] E.G.Friedman, "Clock Distribution Networks in Synchronous Digital Integrated Circuits," *In Proceedings Of the IEEE,*vol.89, No 5, May 2001.

[113] B. Mihajlović and Ž. Žilić, "Real-time address trace compression for emulated and real system-on-chip processor core debugging," In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI (GLSVLSI '11). ACM*, New York, NY,USA,331-336.DOI=10.1145/1973009.1973075.

[114] S. Rigler, "FPGA-Based Lossless Data Compression Using GNU Zip," M.S thesis, Electrical and Computer Engineering Waterloo, Ontario, Canada, 2007.

[115] T. Summers "Compressing Log Files with Hardware Based GZIP," March 29, 2010 , AHA Products Group of Comtech EF Data Corporation.

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
   Ft. Belvoir, Virginia

2. Dudley Knox Library  
   Naval Postgraduate School  
   Monterey, California

3. Chairman  
   Department of Computer Science  
   Naval Postgraduate School  
   Monterey, California

4. Professor Ted Huffmire  
   Department of Computer Science  
   Naval Postgraduate School  
   Monterey, California

5. Professor Timothy. E. Levin  
   Department of Computer Science  
   Naval Postgraduate School  
   Monterey, California

6. Hellenic Navy General Staff  
   Athens, Greece

7. Embassy of Greece  
   Office of Naval Attaché  
   Washington, District of Columbia

8. Dimitrios Megas  
   Naval Postgraduate School  
   Monterey, California

9. Diretoria de Telecomunicações da Marinha  
   Centro, Rio de Janeiro—Brazil

10. Brazilian Naval Commision  
    Washington, DC

11. Kleber Leandro Pizolato Someira  
    Naval Postgraduate School  
    Monterey, California